Concolic Execution Tailored for Hybrid Fuzzing

Insu Yun

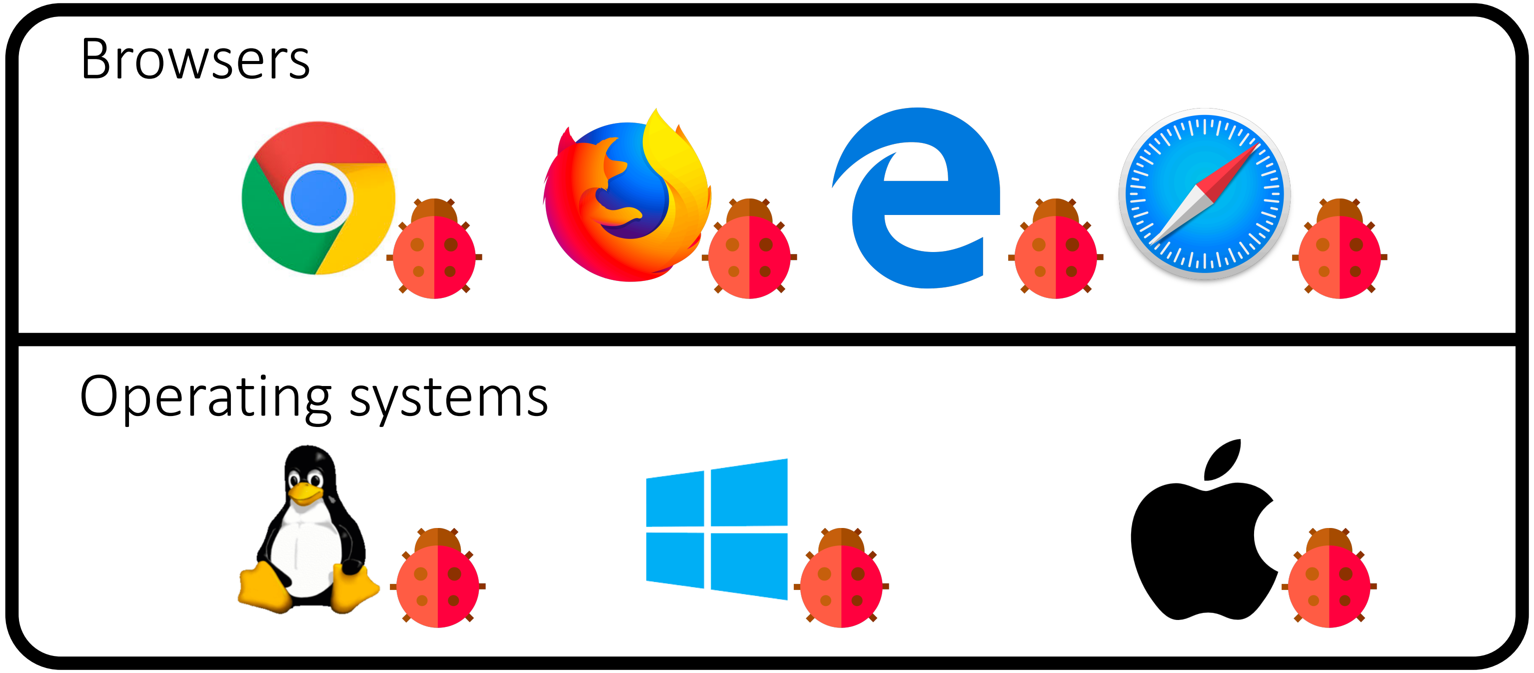
Georgia Institute of Technology

We are using many software applications

Browsers

Operating systems

We are using many (vulnerable) software applications



Attack flow (hacking)

Vulnerability

Attack flow (hacking)

Vulnerability

↓ Exploitation

Attack flow (hacking)

Vulnerability

↓ Exploitation

Compromise

Attack flow (hacking)

Vulnerability

Exploitation

Compromise

Automatic vulnerability finding

APISan (Security ’16), CAB-Fuzz (ATC ‘16),
QSYM (Security’ 18), DIE(Oakland ‘20)
Hybridra(Ongoing)

Attack flow (hacking)

Vulnerability

Exploitation

Compromise

Automatic vulnerability finding

APISan (Security ’16), CAB-Fuzz (ATC ‘16),
QSYM (Security’ 18), DIE(Oakland ‘20)
Hybridra(Ongoing)

Automatic exploitation

ArcHeap (Security ‘20)

Attack flow (hacking)

Vulnerability

Exploitation

Compromise

Automatic vulnerability finding

APISan (Security ’16), CAB-Fuzz (ATC ‘16),
QSYM (Security’ 18), DIE(Oakland ‘20)
Hybridra(Ongoing)

Automatic exploitation

ArcHeap (Security ‘20)

Fixing / Mitigation

HDFI (Oakland ’16), REPT (OSDI ‘18)
FFMalloc (Security’ 20)

Attack flow (hacking)

Vulnerability

Exploitation

Compromise

Automatic vulnerability finding

APISan (Security '16), CAB-Fuzz (ATC '16),

QSYM (Security' 18), DIE(Oakland '20)

Hybridra(Ongoing)

Automatic exploitation

ArcHeap (Security '20)

Fixing / Mitigation

HDFI (Oakland '16), REPT (OSDI '18)

FFMalloc (Security' 20)

Today’s talk

QSYM: A Binary-level Concolic Execution Engine for *Hybrid fuzzing*

- Binary
- User applications

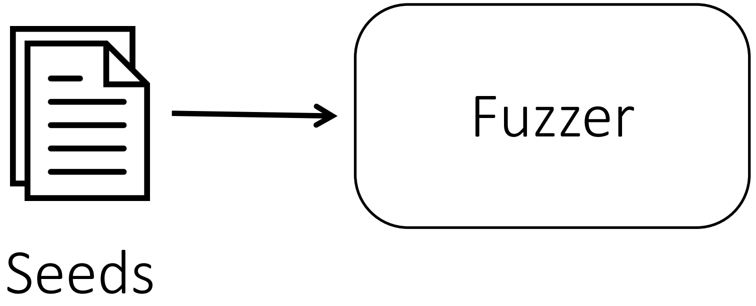
Hybridra: A *Hybrid Fuzzer* for Kernel File Systems

- Source code
- File systems

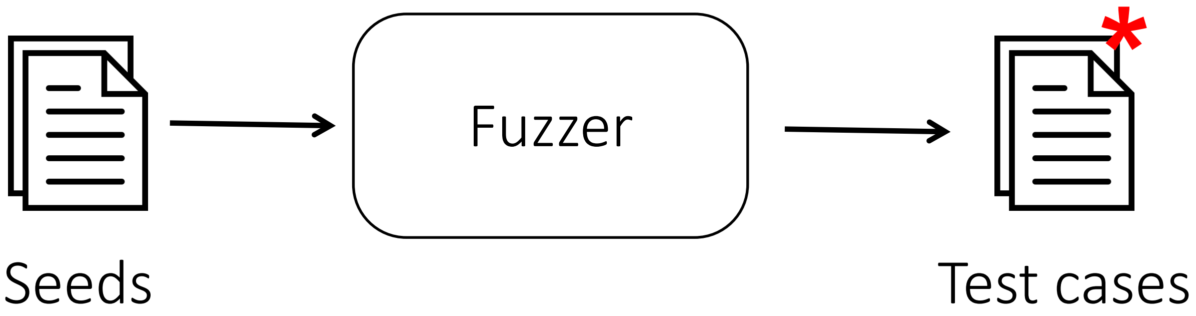
Overview of 40-year-old random testing (fuzzing)

Seeds

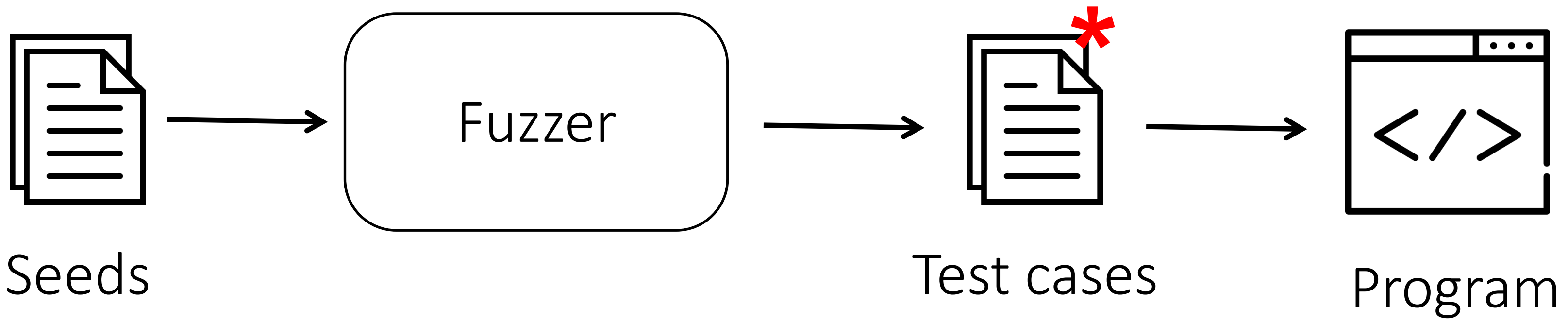
Overview of 40-year-old random testing (fuzzing)



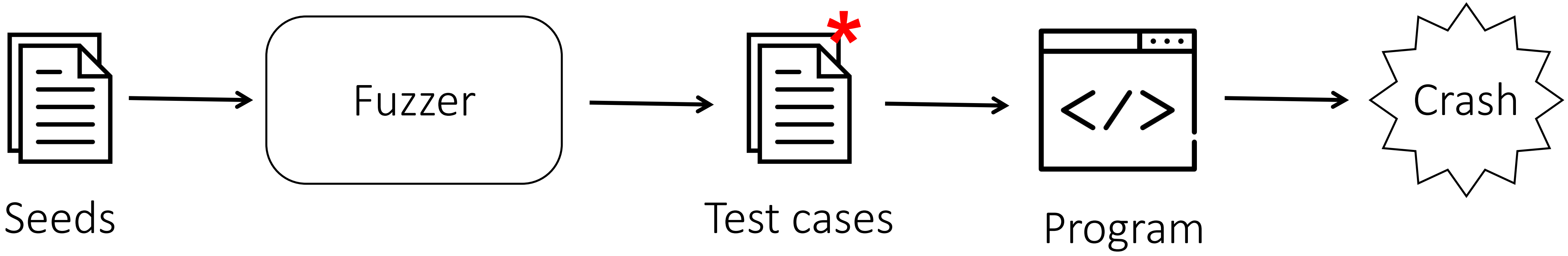
Overview of 40-year-old random testing (fuzzing)



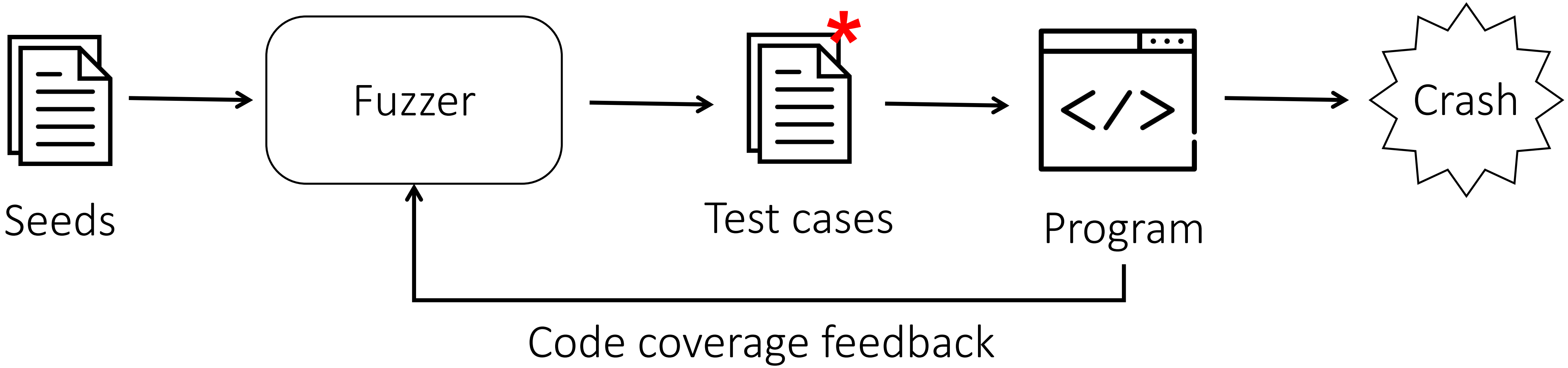
Overview of 40-year-old random testing (fuzzing)



Overview of 40-year-old random testing (fuzzing)



Recent breakthrough: Code coverage feedback



Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Before code coverage feedback,

```
x = input()

if x[0] == 'H’:
    if x[1] == 'A’:
        if x[2] == 'R’:
            if x[3] == 'D’:
                crash()
```

Seeds

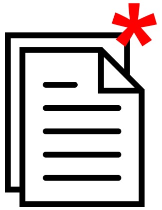
```
x = ‘E4SY'
```

Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = ‘E4SY'



Test cases

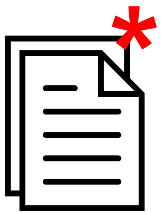
x = ‘E4SI'

Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'



Test cases

x = 'E4SI'

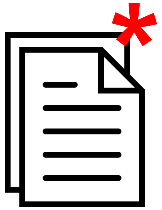
x = 'S4SY'

Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'



Test cases

x = 'E4SI'

x = 'S4SY'

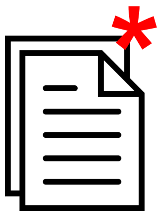
x = 'PTSY'

Before code coverage feedback,

```
x = input()

if x[0] == 'H’:
    if x[1] == 'A’:
        if x[2] == 'R’:
            if x[3] == 'D’:
                crash()
```

Seeds

x = ‘E4SY'



Test cases

x = ‘E4SI'

x = ‘S4SY'

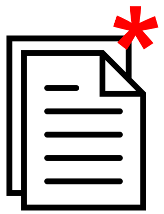
x = ‘PTSY'

x = ‘H4SY'

Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

x = 'E4SY'



x = 'E4SI'

x = 'S4SY'

x = 'PTSY'

x = 'H4SY'

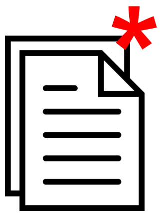
x = 'O4SY'

Before code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'



Test cases

x = 'E4SI'

x = 'S4SY'

x = 'PTSY'

x = 'H4SY'

x = 'O4SY'

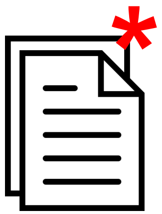
...

Before code coverage feedback,

```
x = input()

if x[0] == 'H’:
    if x[1] == 'A’:
        if x[2] == 'R’:
            if x[3] == 'D’:
                crash()
```

Seeds

x = ‘E4SY'



Test cases

x = ‘E4SI'

x = ‘S4SY'

x = ‘PTSY'

x = ‘H4SY'

x = ‘O4SY'

…

$$P(\text{crash}) = 2^{-32}$$

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = ‘E4SY'

Test cases *

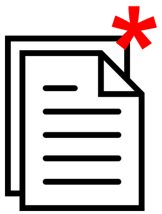
x = ‘E4SI'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = ‘E4SY'



Test cases

x = ‘E4SI'

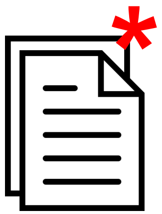
x = ‘S4SY'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = ‘E4SY'



Test cases

x = ‘E4SI'

x = ‘S4SY'

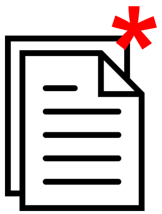
x = ‘ETSY'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
  if x[1] == 'A':
    if x[2] == 'R':
      if x[3] == 'D':
        crash()
```

Seeds

x = 'E4SY'



Test cases

x = 'E4SI'

x = 'S4SY'

x = 'ETSY'

x = 'H4SY'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'

Test cases

x = 'E4SI'

x = 'S4SY'

x = 'ETSY'

x = 'H4SY'

After code coverage feedback,

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'E4SY'

Test cases *

x = 'E4SI'

x = 'S4SY'

x = 'ETSY'

x = 'H4SY'

New code coverage!

Generate test cases from a test case that introduces new code coverage

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'EASY'

x = 'H4SY'

Generate test cases from a test case that introduces new code coverage

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'EASY'

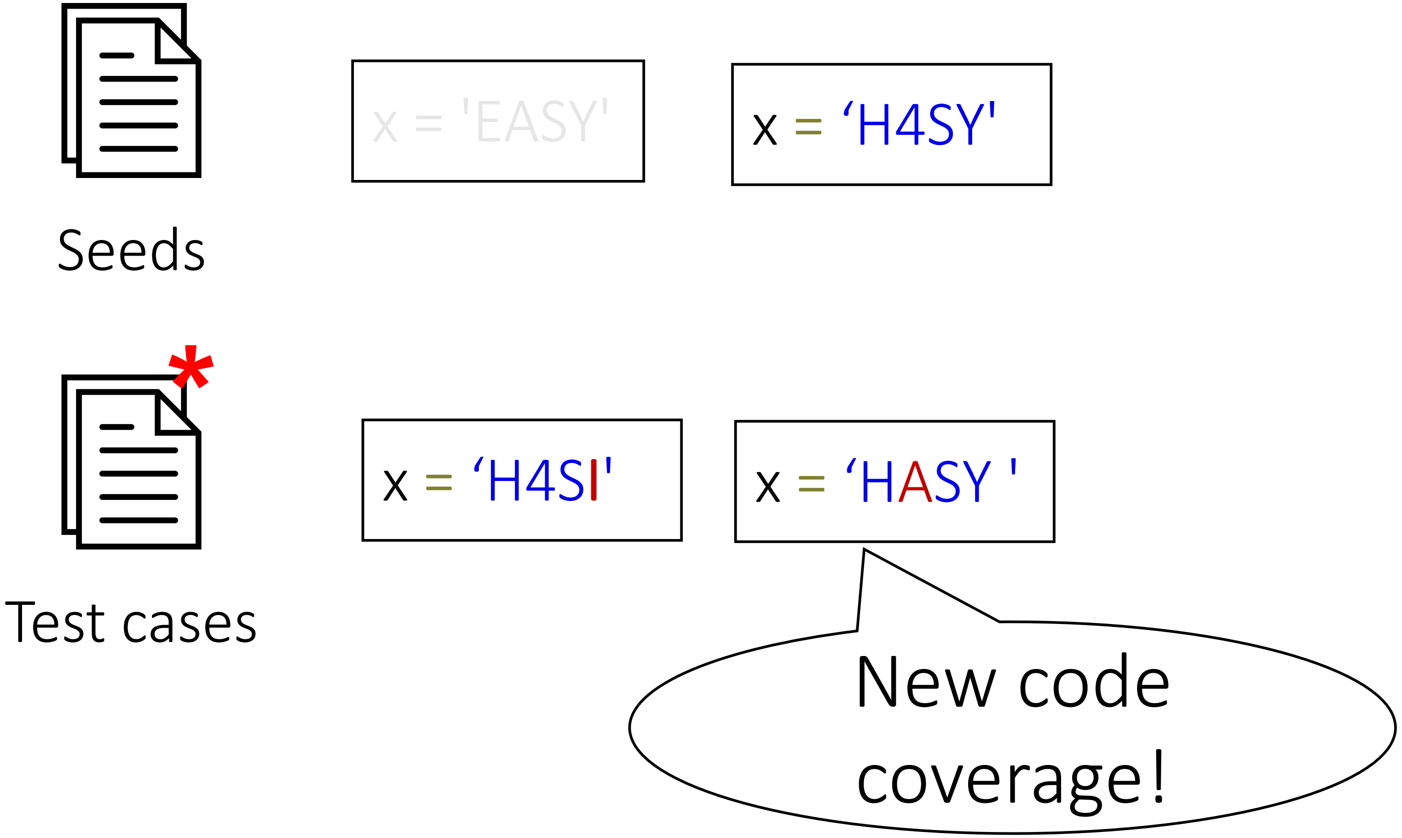
x = 'H4SY'

Test cases *

x = 'H4SI'

Generate test cases from a test case that introduces new code coverage

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```



Generate test cases from a test case that introduces new code coverage

```
x = input()

if x[0] == 'H':
    if x[1] == 'A':
        if x[2] == 'R':
            if x[3] == 'D':
                crash()
```

Seeds

x = 'EASY'

x = 'H4SY'

Test cases *

x = 'H4SI'

x = 'HASY '

New code coverage!

$$P(crash) = 2^{-32}$$

Generate test cases from a test case that introduces new code coverage

```
x = input()

if x[0] == 'H’:
    if x[1] == 'A’:
        if x[2] == 'R’:
            if x[3] == 'D’:
                crash()
```

Seeds

x = 'EASY'

x = ‘H4SY'

Test cases *

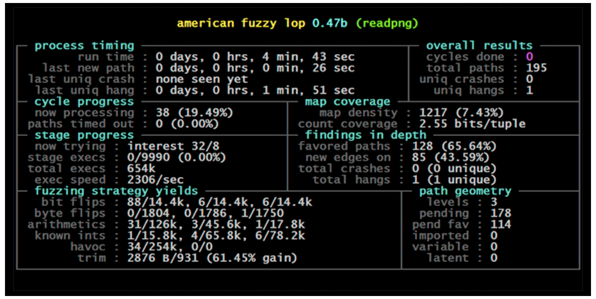
x = ‘H4SI'

x = ‘HASY '

New code coverage!

$$P(crash) = \cancel{2^{-32}} = \underbrace{2^{-8}}_{\text{Per-byte}} \times \underbrace{2^{-2}}_{\text{4 bytes}} = 2^{-10}$$

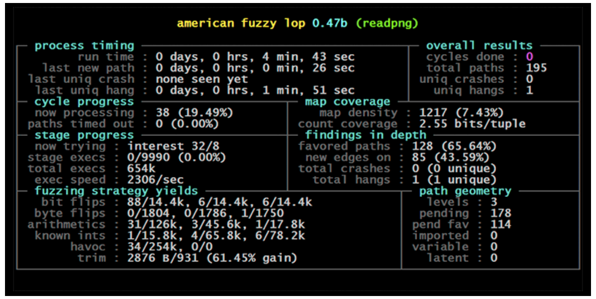
Coverage-guided fuzzing is effective



AFL

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs
  e.g.,) Safari, Firefox, OpenSSL, …

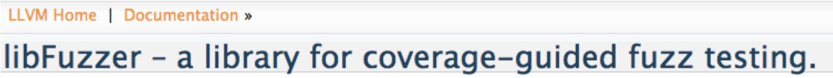
Coverage-guided fuzzing is effective



AFL

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs
  e.g.,) Safari, Firefox, OpenSSL, …

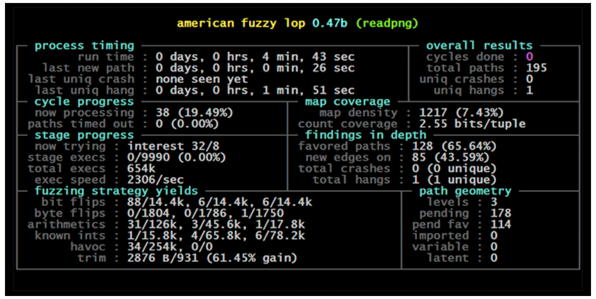
LLVM COMPILER INFRASTRUCTURE



libFuzzer

- LLVM community developed
- A library to include random testing as a part of projects
  e.g.,) LLVM, Chromium, Tensorflow, …

Coverage-guided fuzzing is effective



AFL

LLVM Home | Documentation »

libFuzzer – a library for coverage-guided fuzz testing.

libFuzzer

OSS-Fuzz

- Fuzzer developed by Google
- Re-discover coverage-guided fuzzing
- Found hundreds of bugs in many programs
  e.g.,) Safari, Firefox, OpenSSL, …

- LLVM community developed
- A library to include random testing as a part of projects
  e.g.,) LLVM, Chromium, Tensorflow, …

- Use Google's cloud resources to fuzz open-source software
- 4 trillion test cases a week

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678^2
if x * x == 459684 :
    crash()
```

Seeds

```
x = 0
```

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```

Seeds

x = 0

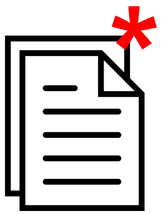
Test cases

x = 3

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
  crash()
```

x = 0



x = 3

x = 452

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
  crash()
```

Seeds

x = 0

Test cases *

x = 3

x = 452

x = 942

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
  crash()
```

Seeds

x = 0

Test cases *

x = 3

x = 452

x = 942

x = 512

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```

Seeds

x = 0

Test cases

x = 3

x = 452

x = 942

x = 512

x = 28

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
  crash()
```

Seeds

x = 0

Test cases

x = 3

x = 452

x = 942

x = 512

x = 28

...

Limitations of fuzzing: Randomly hard-to-find

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
  crash()
```

Seeds

x = 0

Test cases

x = 3

x = 452

x = 942

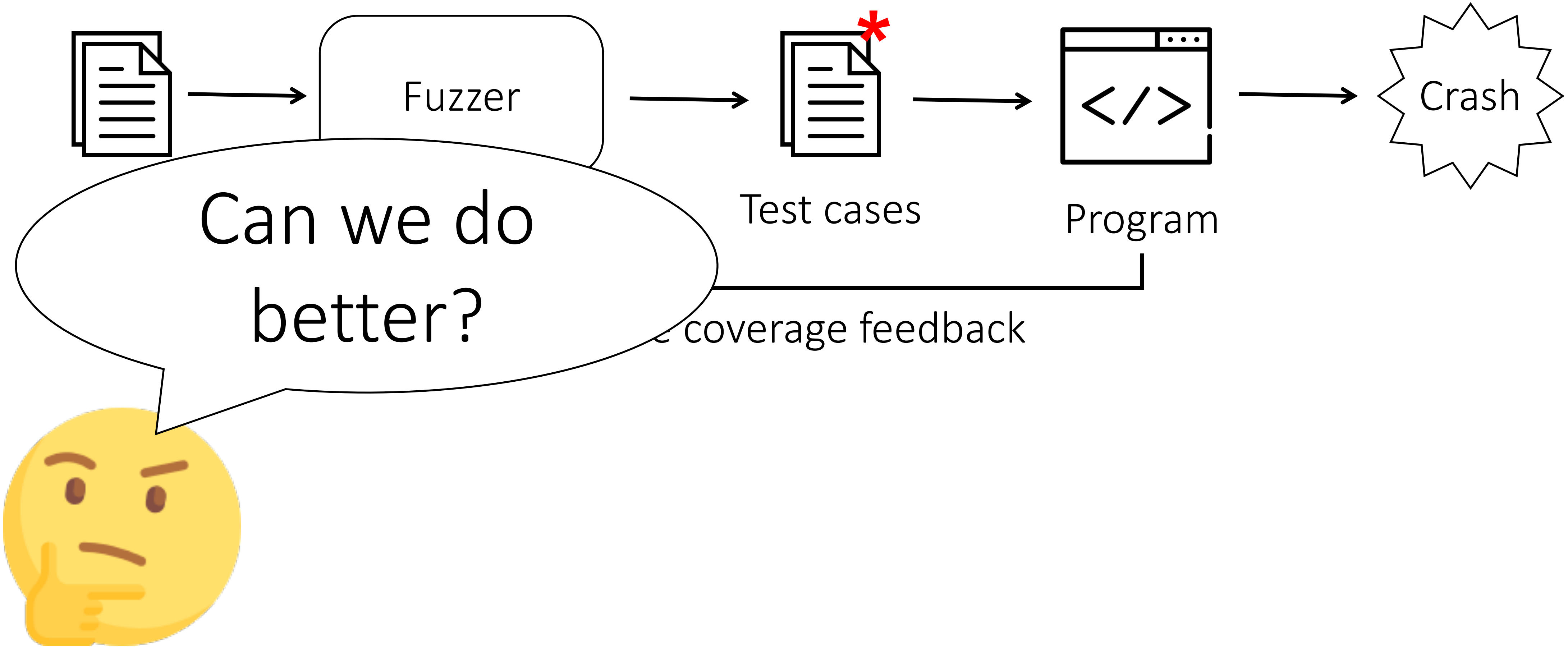
x = 512

x = 28

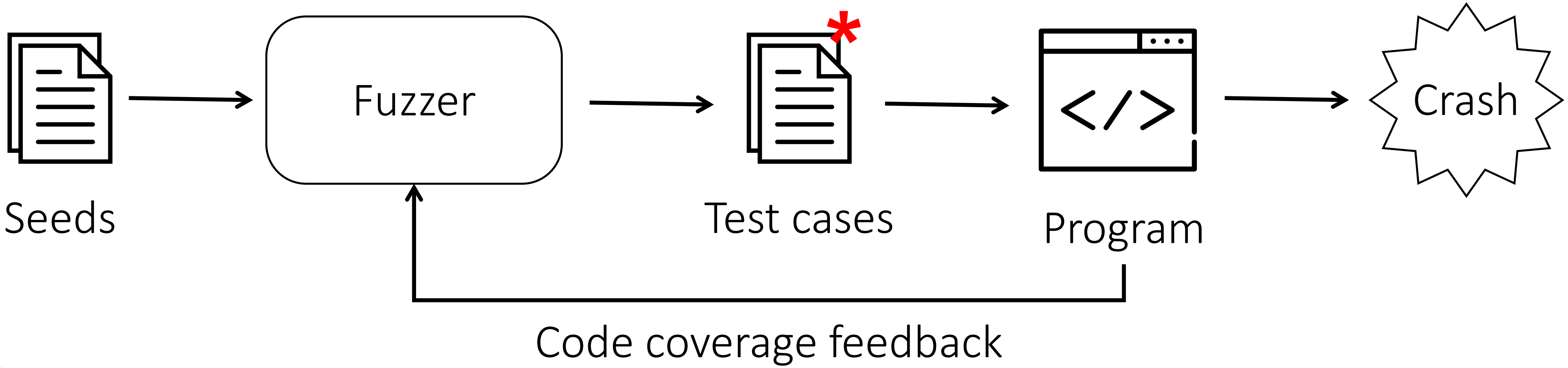
...

$$P(\text{crash}) = 2^{-32}$$

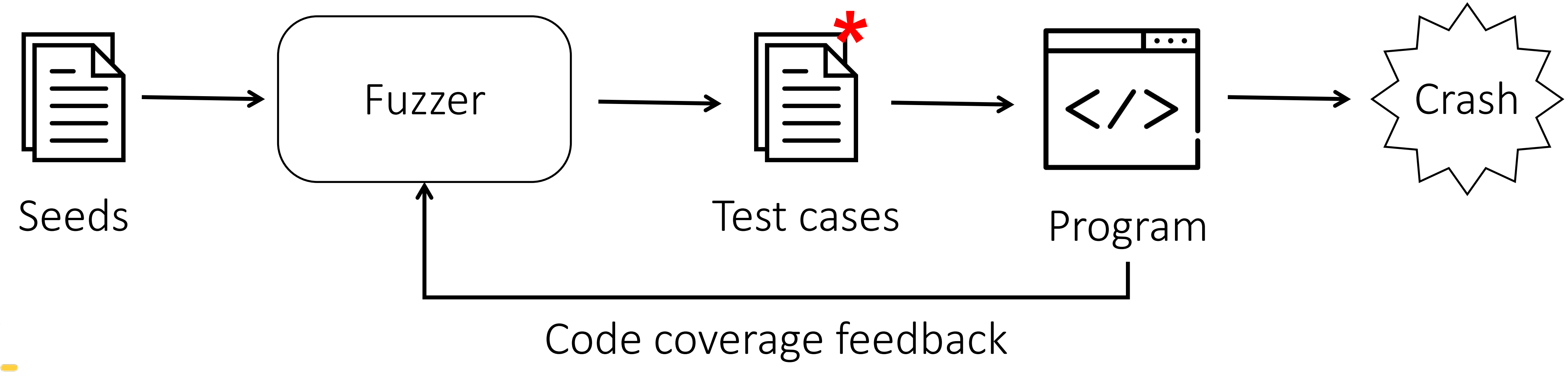
Limitations of coverage-guided fuzzing

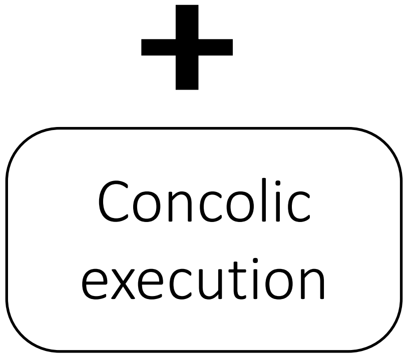


Limitations of coverage-guided fuzzing

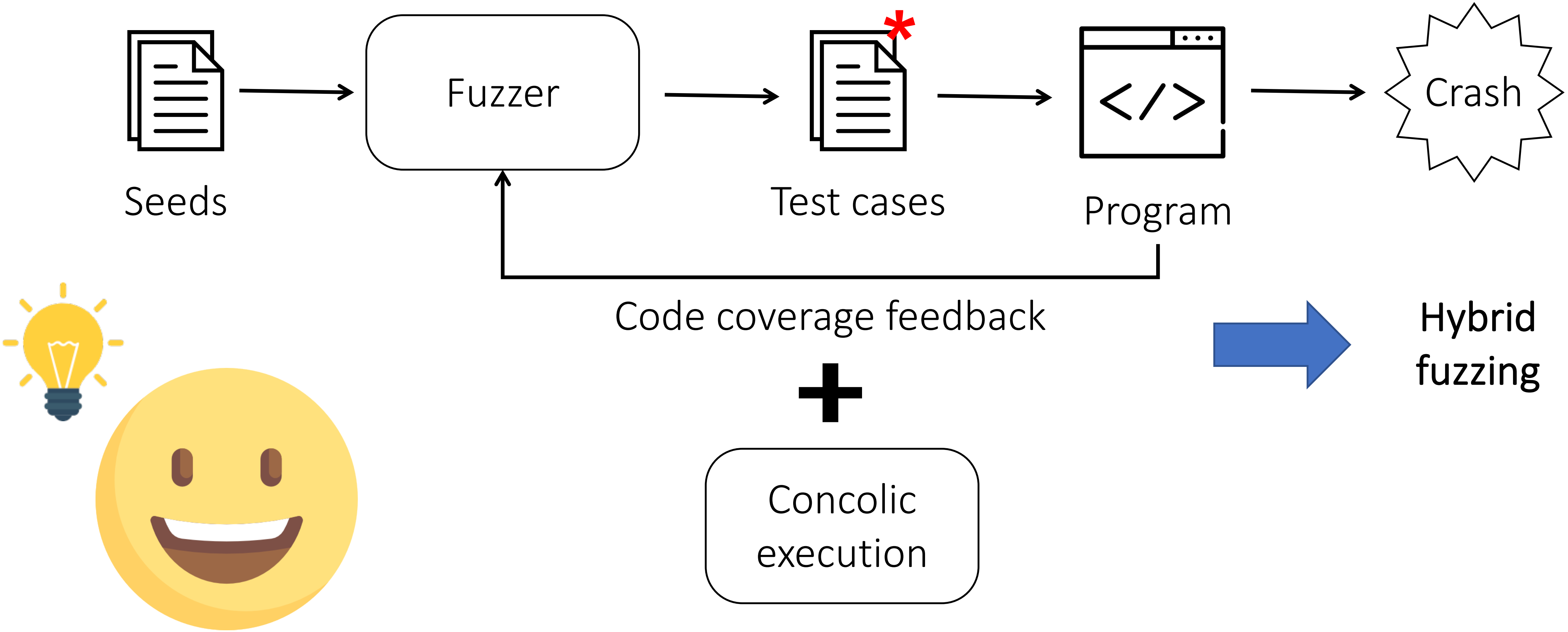


Limitations of coverage-guided fuzzing





Limitations of coverage-guided fuzzing



Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
   crash()
```

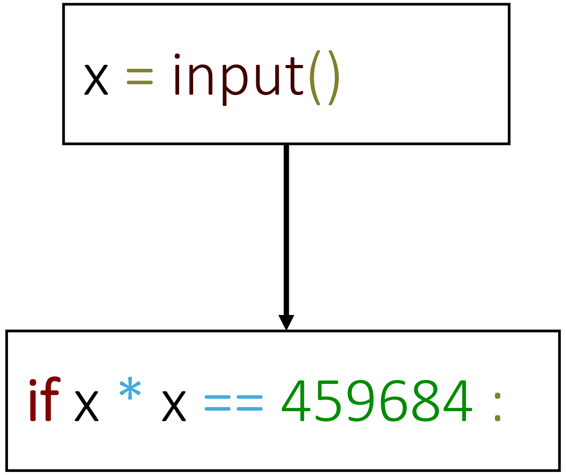
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```

```
x = input()
```

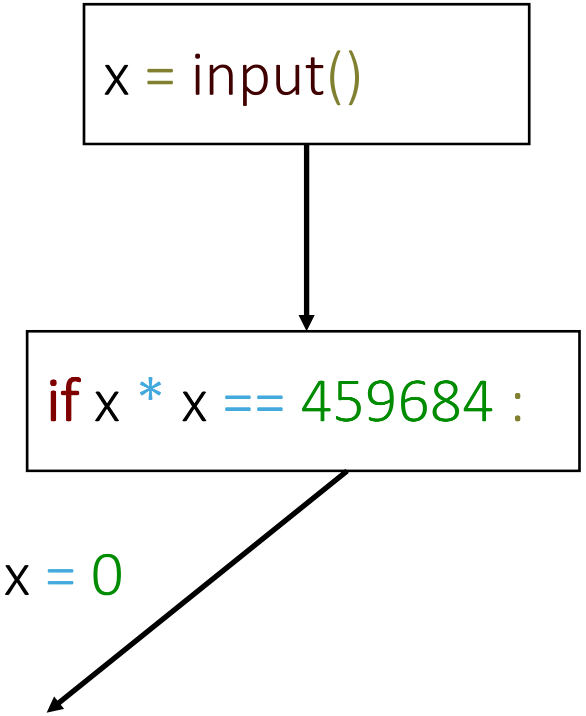
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```



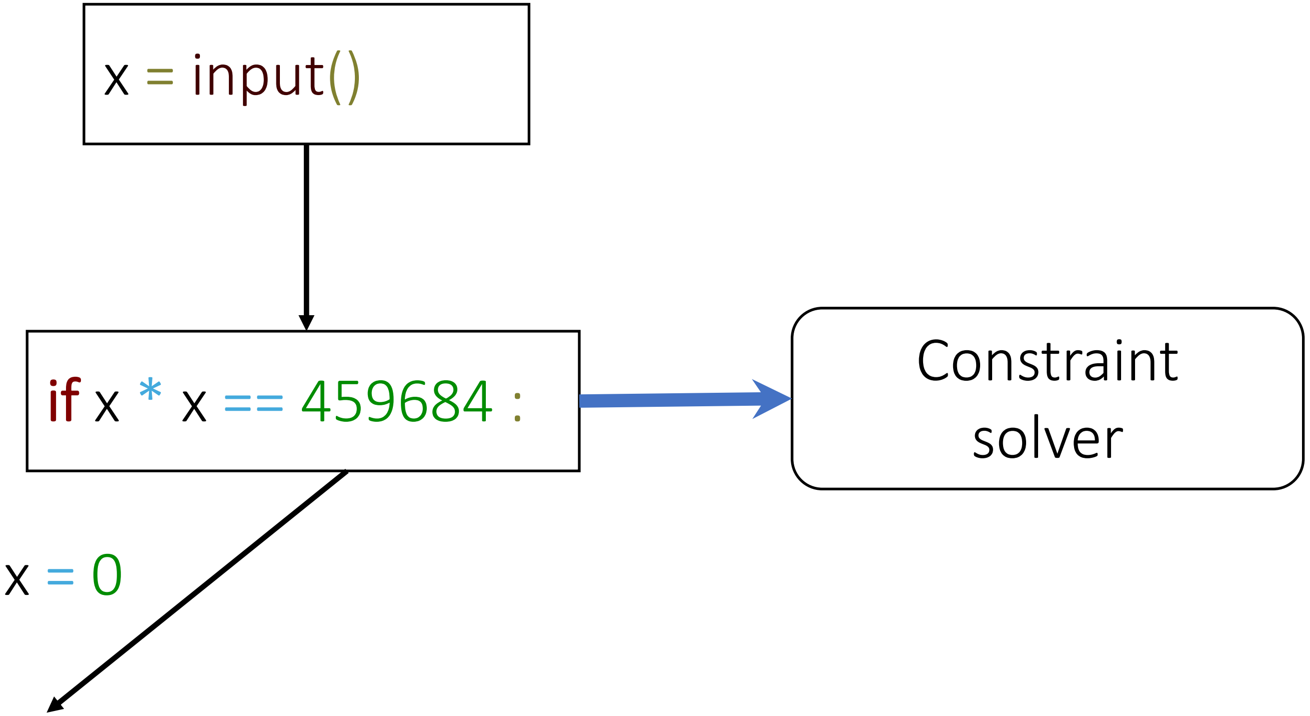
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```



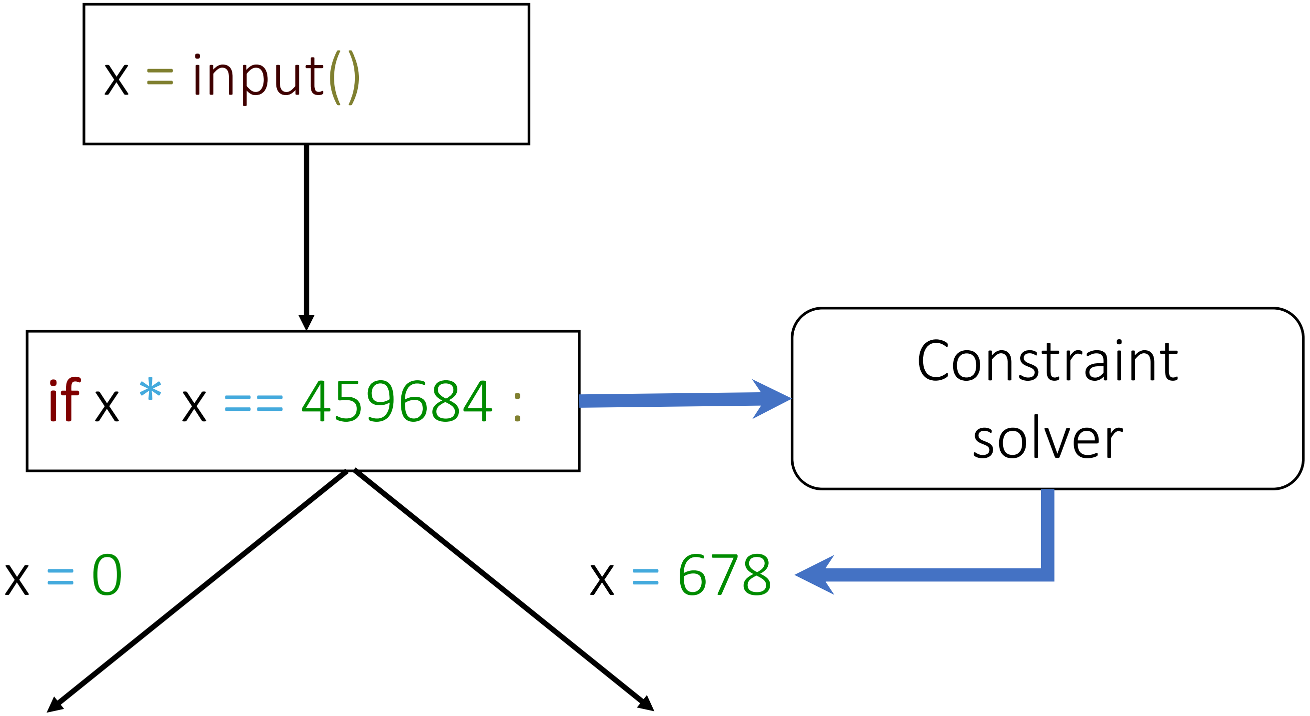
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```



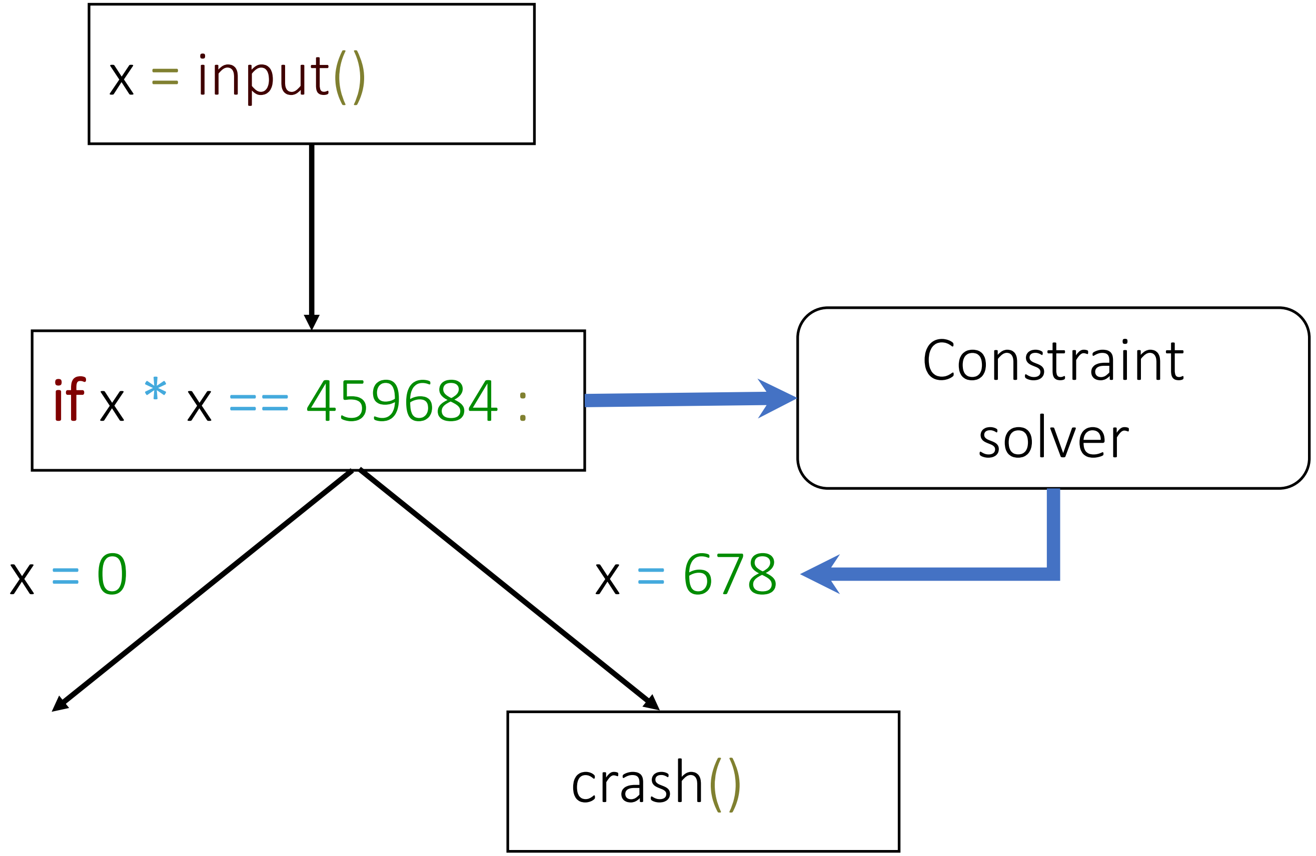
Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
    crash()
```



Concolic execution can help fuzzing by finding hard-to-find test cases

```
x = int(input())

// 459684 == 678²
if x * x == 459684 :
   crash()
```



Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

- Organized by DARPA in 2017
- Build a system to find bugs, exploit and patch on binaries

- Over 100 teams → 7 teams were qualified (include our team)
- Almost $4 million for prize money

Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

- Organized by DARPA in 2017
- Build a system to find bugs, exploit and patch on binaries

- Over 100 teams → 7 teams were qualified (include our team)
- Almost $4 million for prize money

- *Small binaries: a few KB*

Hybrid fuzzing has achieved great success in small-scale study (DARPA Cyber Grand Challenge)

- The winner from CMU used hybrid fuzzing

- Shellphish open-sourced their tool, Driller
  - Won 3rd place in CGC competition
  - Found 6 new crashes: cannot be found by fuzzing or concolic execution

But, hybrid fuzzing fails to scale real-world applications

Cannot find **ANY** bug in real-world software using Driller!

Current concolic executors suffer several problems to be used in hybrid fuzzing

Generating constraints is too slow

Current concolic executors suffer several problems to be used in hybrid fuzzing

Generating constraints is too slow

Not effective in generating test cases

Current concolic executors suffer several problems to be used in hybrid fuzzing

Generating constraints is too slow

Not effective in generating test cases

Symbolic emulation is well-known to be much slower than concrete execution

```
int is_double(int* a, int b) {
   return *a == 2 * b;
}
```

Symbolic emulation is well-known to be much slower than concrete execution

Concrete Execution:
Just Execute!

```
int is_double(int* a, int b) {
    return *a == 2 * b;
}
```

Symbolic emulation is well-known to be much slower than concrete execution

```
int is_double(int* a, int b) {
   return *a == 2 * b;
}
```

Symbolic Emulation:
Taint tracking

Symbolic emulation is well-known to be much slower than concrete execution

```c
int is_double(int* a, int b) {
   return *a == 2 * b;
}
```

Symbolic Emulation:
Taint tracking

Symbolic Emulation:
Constructing symbols
(e.g., *a == 2 * Symbol_b)

Symbolic emulation is well-known to be much slower than concrete execution

```
int is_double(int* a, int b) {
   return *a == 2 * b;
}
```

Symbolic Emulation:
Taint tracking

Symbolic Emulation:
Constructing symbols
(e.g., *a == 2 * $Symbol_b$)

Symbolic Emulation:
Interpretation
(e.g., *a = ???)

Symbolic emulation is well-known to be much slower than concrete execution

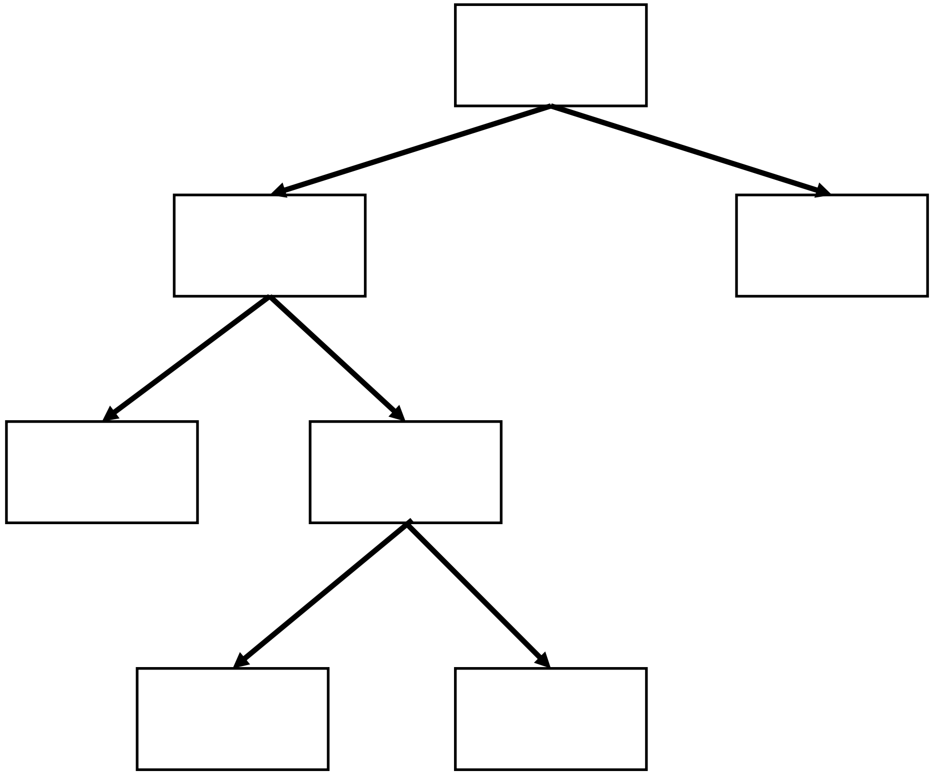
Symbolic Emulation:
Taint tracking

Symbolic emulation is 1,000x slower than concrete execution!

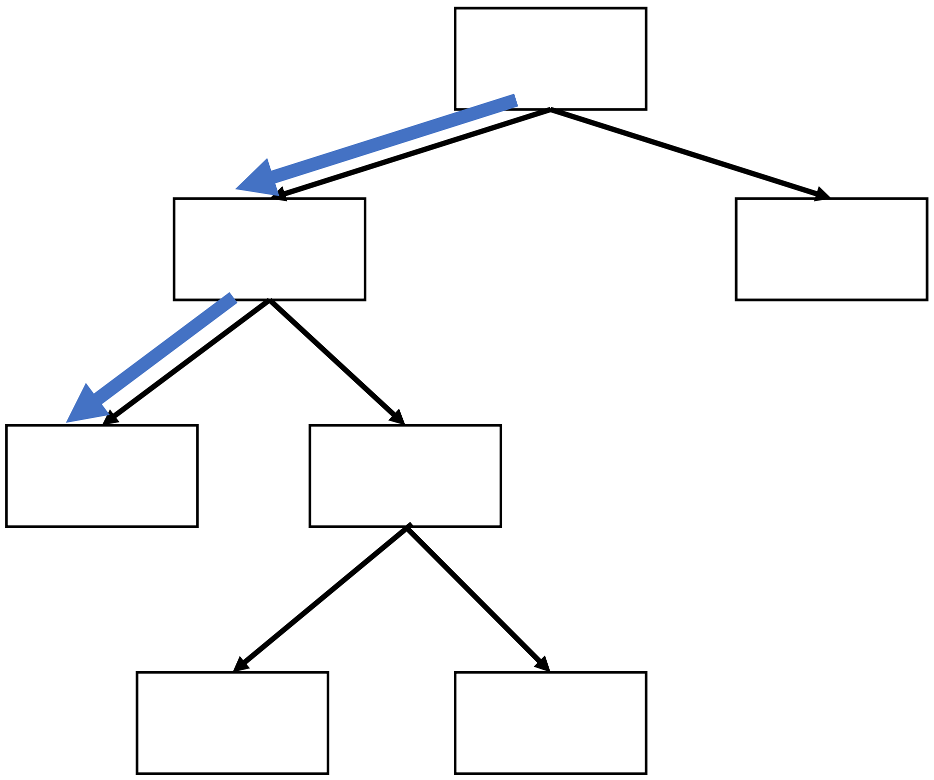
Symbolic Emulation:
Constructing symbols
(e.g., *a == 2 * $Symbol_b$)

Symbolic Emulation:
Interpretation
(e.g., *a = ???)

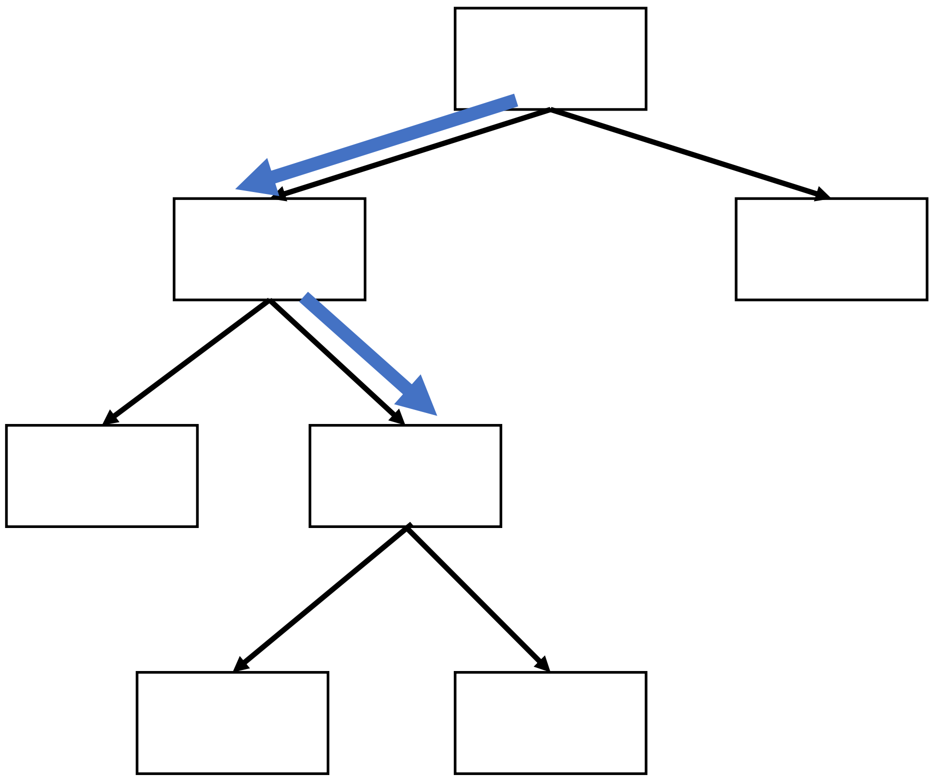
State forking can be used to solve this problem



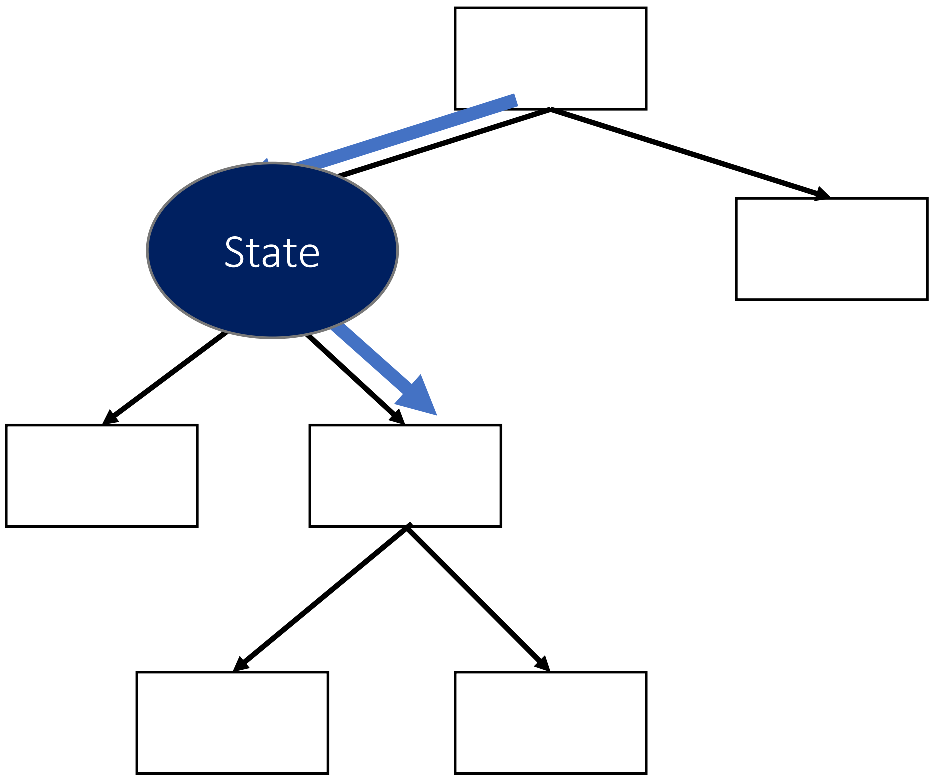
State forking can be used to solve this problem



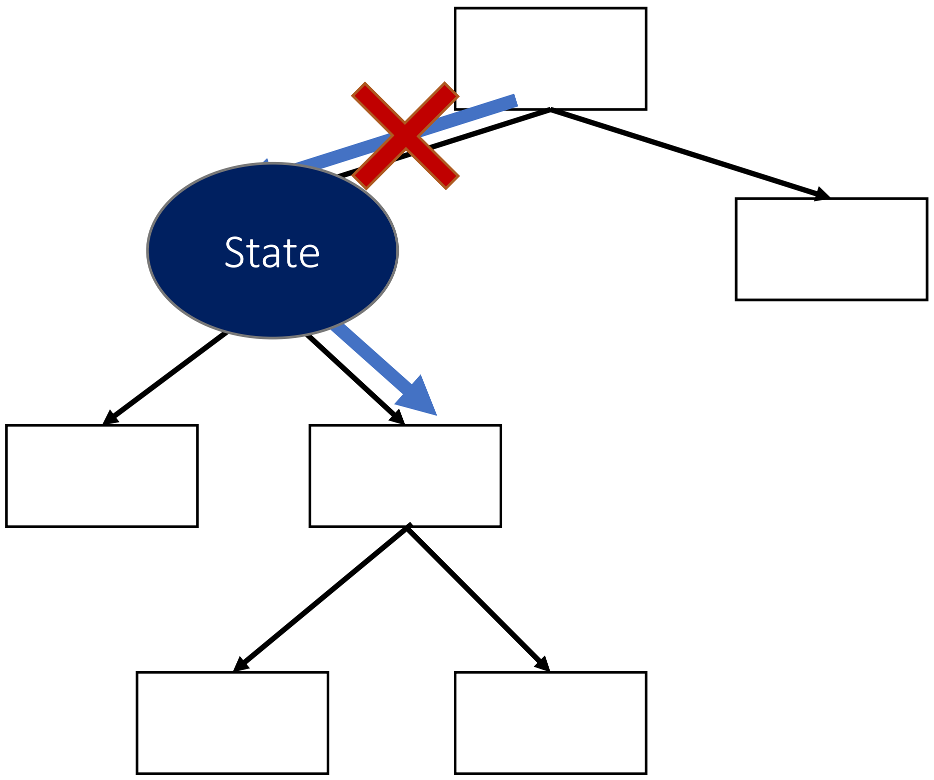
State forking can be used to solve this problem



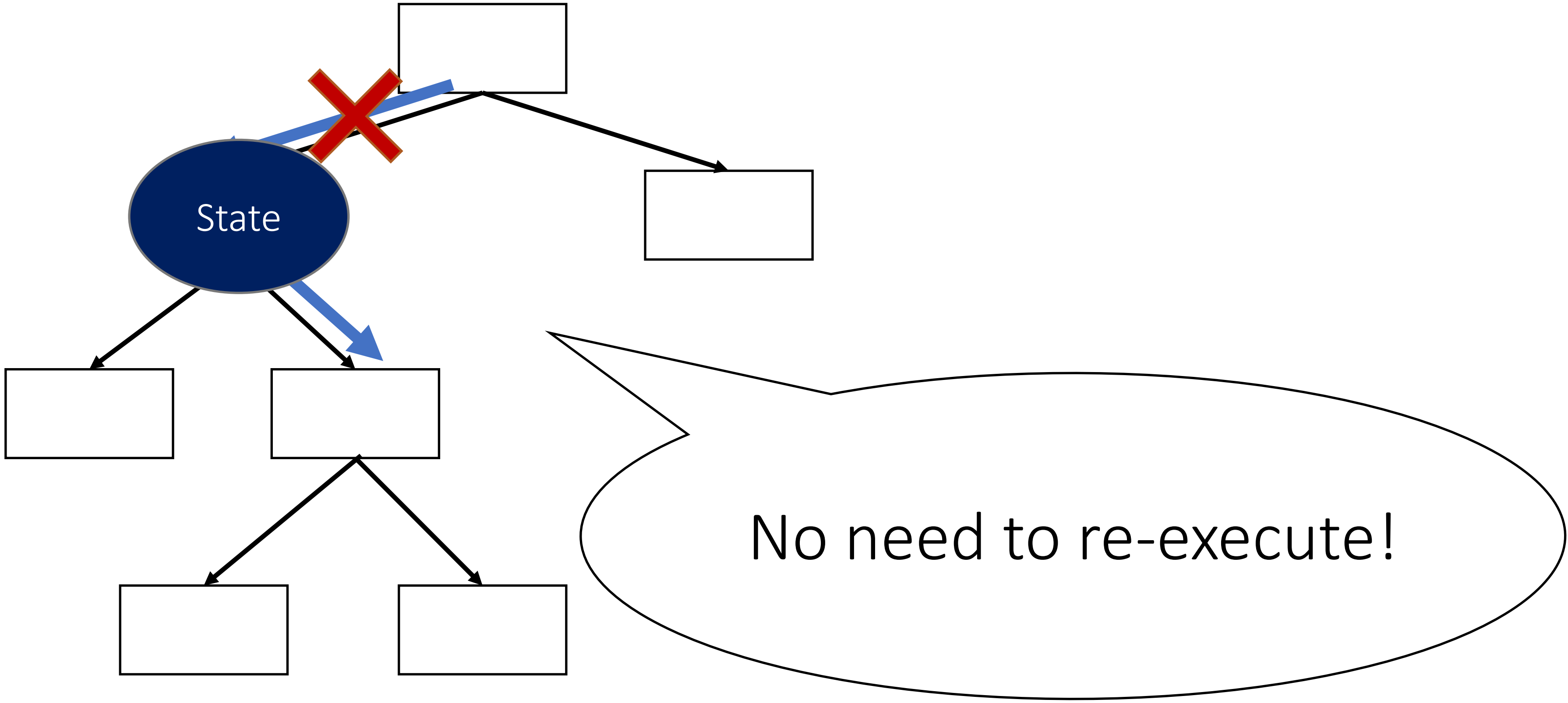
State forking can be used to solve this problem



State forking can be used to solve this problem



State forking can be used to solve this problem



State forking is limited in hybrid fuzzing

State forking is limited in hybrid fuzzing

- # of states is enormous in a complex real-world program
  => Large performance overhead

- Hybrid fuzzing explores paths randomly following fuzzing
  => Low reusability

State forking is limited in hybrid fuzzing

- # of states is enormous in a complex real-world program
  => Large performance overhead

- Hybrid fuzzing explores paths randomly following fuzzing
  => Low reusability

State forking cannot help slow symbolic emulation in hybrid fuzzing!

Current concolic executors have several problems to be used in hybrid fuzzing

Generating constraints is too slow

Not effective in generating test cases

Completeness of concolic execution often blocks its further exploration

```
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Completeness of concolic execution often blocks its further exploration

```
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Analyze every routine for completeness!

Soundness of concolic execution incurs unnecessary re-execution

```
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
  if (x == 0)
    do_something();

  if (x * x == 1234 * 1234)
    crash();
}
```

Soundness of concolic execution incurs unnecessary re-execution

```
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
  if (x == 0)
    do_something();

  if (x * x == 1234 * 1234)
    crash();
}
```

Soundness of concolic execution incurs unnecessary re-execution

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
  if (x == 0)
    do_something();

  if (x * x == 1234 * 1234)
    crash();
}
```

Unsatisfiable!

Soundness of concolic execution incurs unnecessary re-execution

```c
// 'x' is symbolic and 'x' == 0 in a given input
int soundess(int x) {
  if (x == 0)
    do_something();

  if (x * x == 1234 * 1234)
    crash();
}
```

Unsatisfiable!

Don’t make possibly incorrect test cases for soundness!

Our approach

Generating constraints is too slow

Not effective in generating test cases

Our approach

Generating constraints is too slow

Not effective in generating test cases

Systematic approach for fast symbolic emulation

Our approach

Generating constraints is too slow → Systematic approach for fast symbolic emulation

Not effective in generating test cases → New heuristics for hybrid fuzzing

Our approach: QSYM

QSYM

Systematic approach for fast symbolic emulation	Instruction-level concolic execution (For binary)
New heuristics for hybrid fuzzing	Optimistic solving and basic block pruning

Our approach: Hybridra

Hybridra

Systematic approach for fast symbolic emulation	Compilation-based concolic execution (For source code)
New heuristics for hybrid fuzzing	Staged reduction + Heuristics from QSYM

Related work: Whitebox fuzzing

- Goal
  - Hybrid fuzzing: Make a test case for fuzzing
  - Whitebox fuzzing: Explore a program state solely

- Exploration
  - Hybrid fuzzing: Random
  - Whitebox fuzzing: Systematic

- Strategy: Hybrid fuzzing’s can be more aggressive thanks to coverage-guided fuzzing (e.g., optimistic solving)

Today's talk

QSYM: A Binary-level Concolic Execution Engine for Hybrid fuzzing

- Binary
- User applications

Hybridra: A Hybrid Fuzzer for Kernel File Systems

- Source code
- File systems

Our system, QSYM, addresses these issues by introducing several key ideas

Problem		Solution
Generating constraints is too slow	→	Instruction-level concolic execution (For binary)
Not effective in generating test cases	→	Optimistic solving and basic block pruning

Our system, QSYM, addresses these issues by introducing several key ideas

Generating constraints is too slow	Instruction-level concolic execution (For binary)
Not effective in generating test cases	Optimistic solving and basic block pruning

QSYM has made several design decisions for improving performance

- Discarding intermediate layer
- Instruction-level symbolic execution

QSYM has made several design decisions for improving performance

- Discarding intermediate layer

Simple, but
BIG
design decision

- Instruction-level symbolic execution

Hybrid fuzzing in a closer look

push ebp
...

Assembly

Hybrid fuzzing in a closer look

```
push ebp
...
```

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

Hybrid fuzzing in a closer look

push ebp
...

Assembly

t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...

Intermediate Representation (IR)

a == 0x1337
∧ b * c == 0xc001
...
∧ z == 0xc0de

Constraints

Hybrid fuzzing in a closer look

```
push ebp
...
```

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

a == 0x1337
∧ b * c == 0xc001
...
∧ z == 0xc0de

Constraints

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Hybrid fuzzing in a closer look

push ebp
...

Assembly

→

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

→

a == 0x1337
∧ b * c == 0xc001
...
∧ z == 0xc0de

Constraints

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck

Problems of IR: The number of instructions increase

```
push ebp
...
```

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

4.96x increase on average!

Problems of IR: Slow transformation speed

push ebp
...

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

Slow...

Problems of IR: Slow transformation speed

```
push ebp
...
```

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

Slow…

Cache it!

Side effects of caching: Basic-block granularity

Side effects of caching: Basic-block granularity

- Cache lookup is also slow

- Use basic-block granularity for caching
    - i.e., transform a basic block into IR and cache

- Unfortunately, ***30%*** of instructions in a basic block are symbolic
  → 70% of instructions are executed without need

Side effects of caching: Basic-block granularity

- Cache lookup is also slow

- Use basic-block granularity for caching
  - i.e., transform a basic block into IR and cache

- Unfortunately, ***30%*** of instructions in a basic block are symbolic
  → 70% of instructions are executed without need

How to solve this challenge?

push ebp
...

Assembly

```
t0 = GET:I32(ebp)
t1 = GET:I32(esp)
t2 = Sub32(t1,0x00000004)
PUT(esp) = t2
STle(t2) = t0
...
```

Intermediate Representation (IR)

a == 0x1337
∧ b * c == 0xc001
...
∧ z == 0xc0de

Constraints

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck

How to solve this c

Performance is

the most important!

push ebp
…

Assembly

t0 = GET:I32(
t1 = GE 2(es
t2 = Sub32(x00 00004)
PUT(esp) 2
STle(t2) t0
…

Intermediate
Representation (IR)

0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

Good: Simplifying implementations
e.g., 981 in x86 vs 115 in VEX

Bad: Performance bottleneck

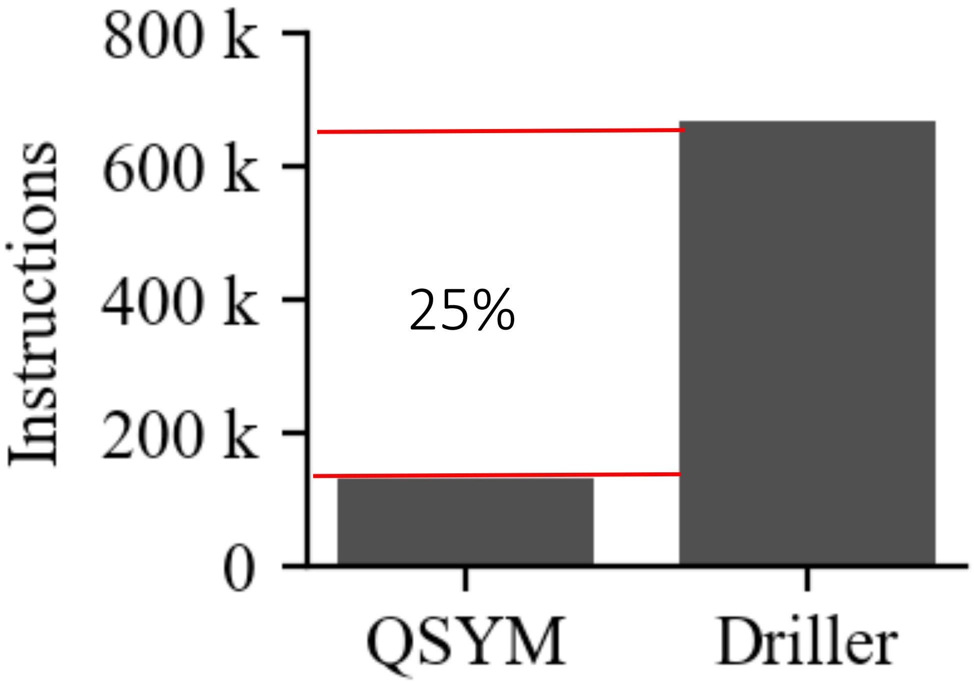
This is a non-trivial job (LoC)

4.7K	13K
Driller	QSYM

https://github.com/angr/angr/tree/master/angr/engines

QSYM reduces the number of symbolically executed instructions

- 126 CGC binaries



2.5x end-to-end performance Improvement

Our system, QSYM, addresses these issues by introducing several key ideas

Generating constraints is too slow → Instruction-level concolic execution (For binary)

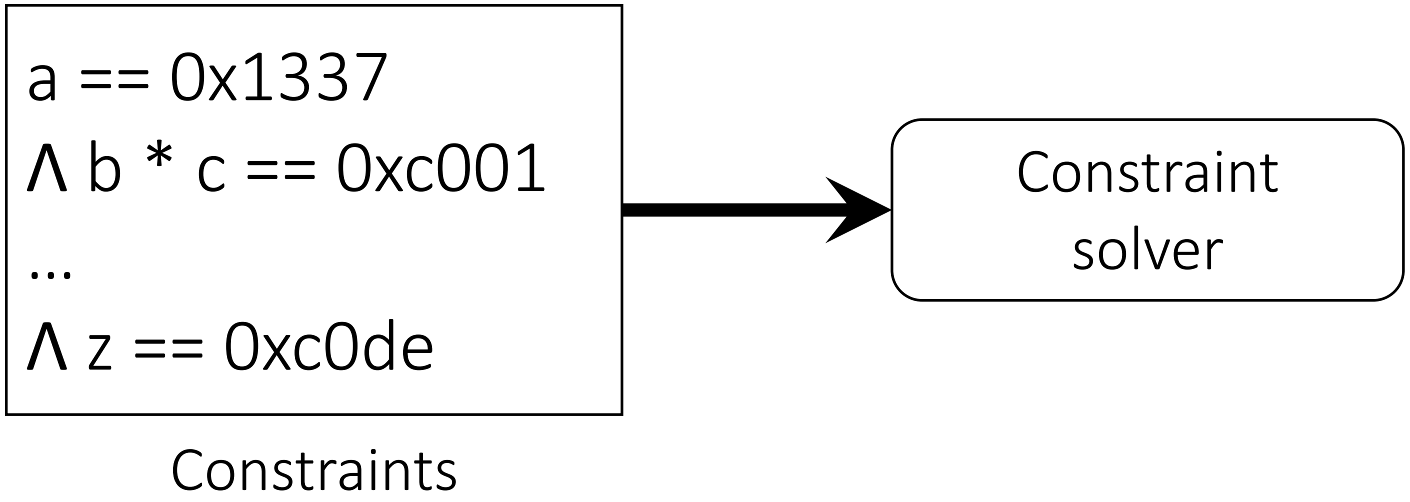
Not effective in generating test cases → Optimistic solving and basic block pruning

Constraint solving can generate a test case that meets given constraints

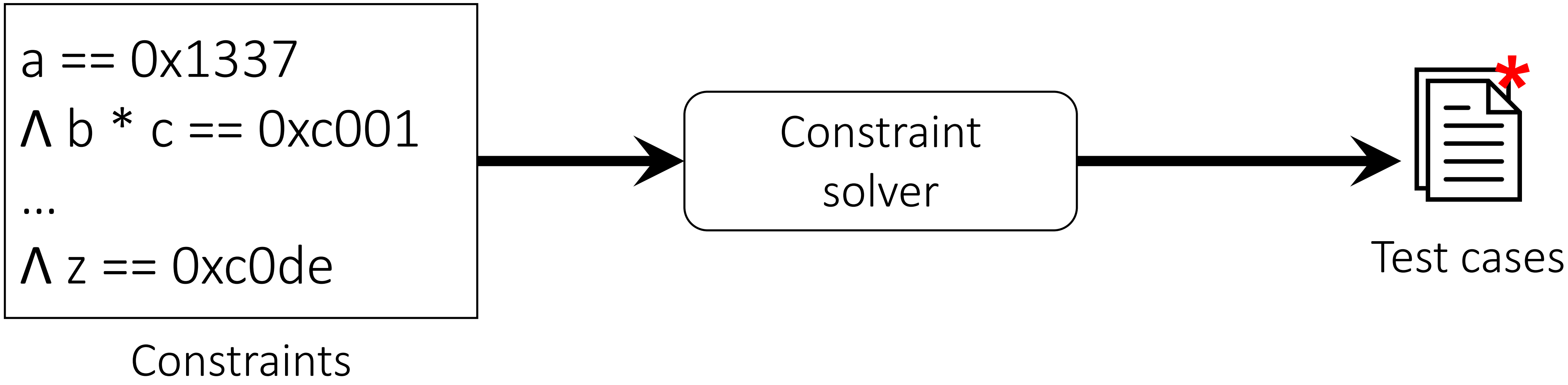
```
a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de
```

Constraints

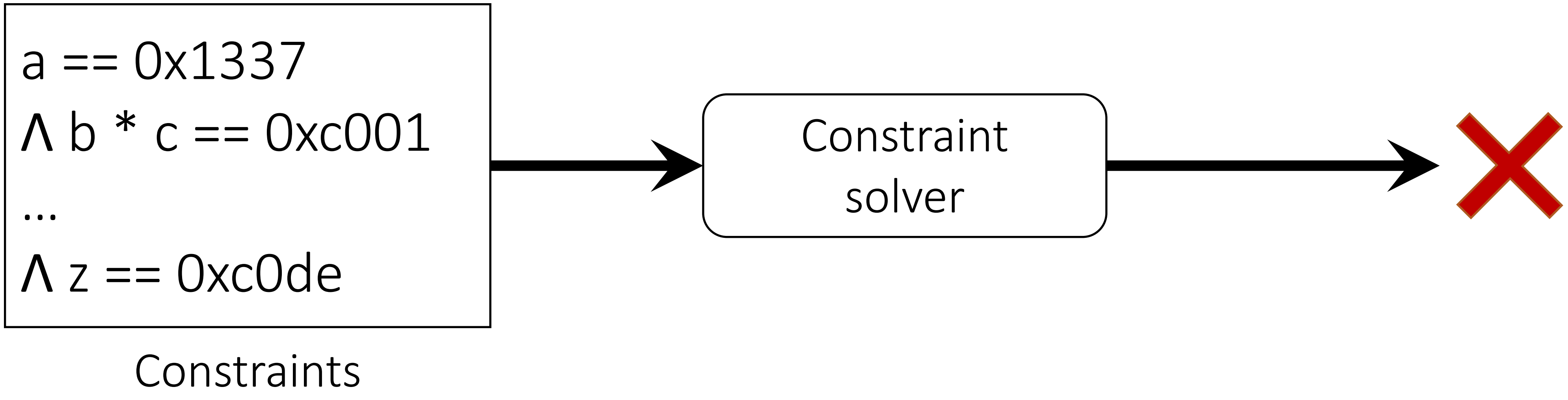
Constraint solving can generate a test case that meets given constraints



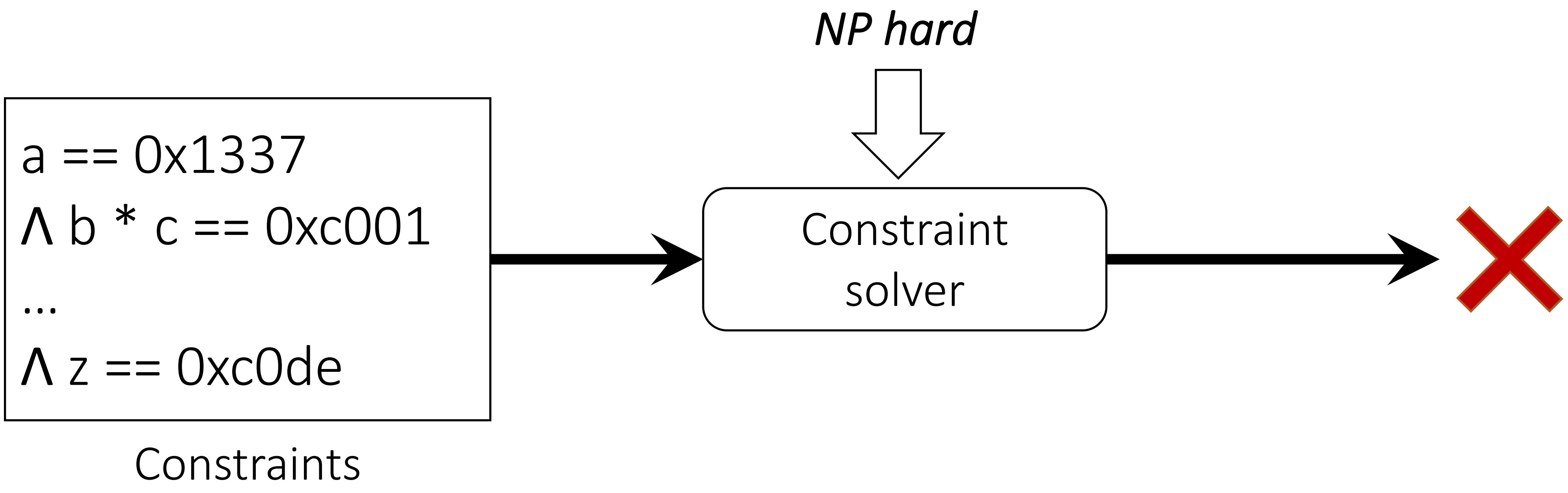
Constraint solving can generate a test case that meets given constraints



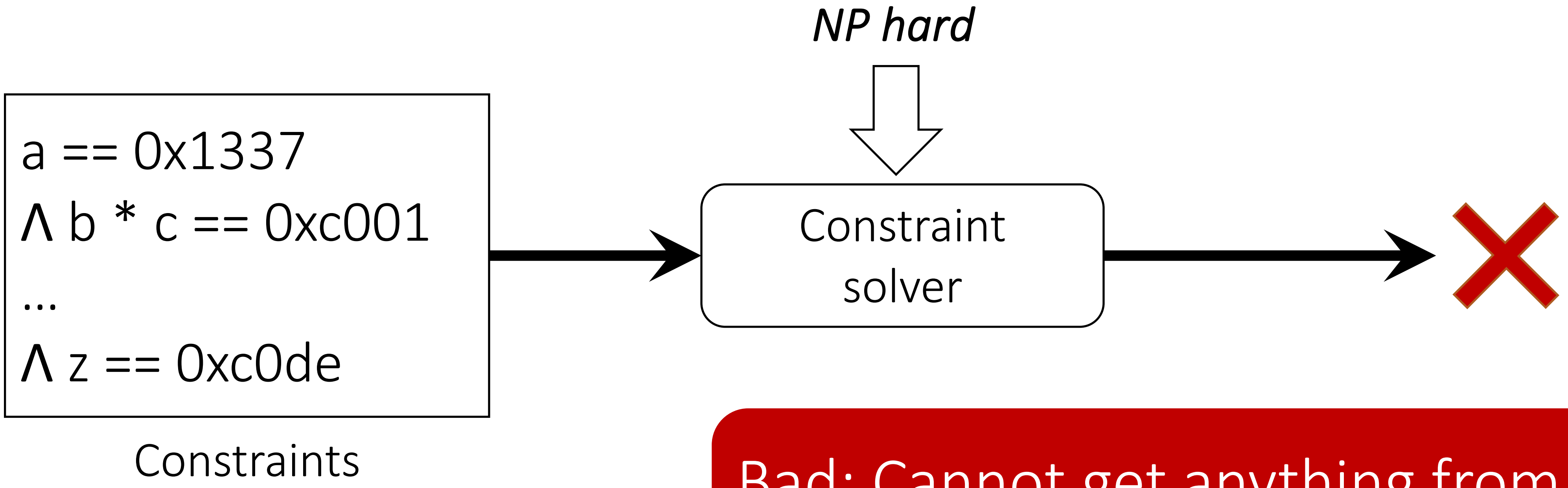
Constraint solving **CANNOT** generate a test case that meets given constraints



Constraint solving **CANNOT** generate a test case that meets given constraints



Constraint solving **CANNOT** generate a test case that meets given constraints



Bad: Cannot get anything from concolic execution

QSYM solves partial constraints to find some test cases

a == 0x1337
∧ b * c == 0xc001
...
$\boxed{\land\ z == \text{0xc0de}}$

QSYM solves partial constraints to find some test cases

a == 0x1337
∧ b * c == 0xc001
...
∧ z == 0xc0de

Constraint solver

Test cases

Good: Can get test cases from concolic execution

QSYM solves partial constraints to find some test cases

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0d

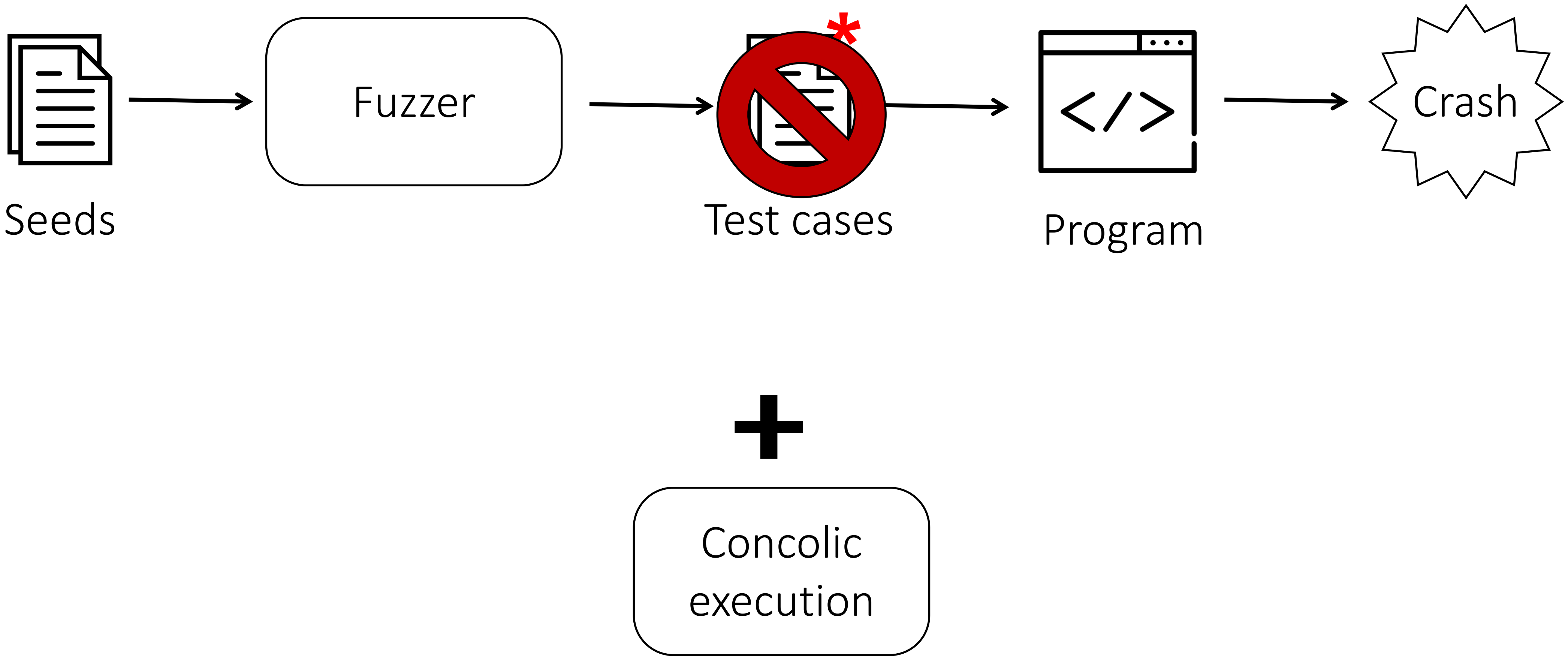
Constraint solver

Test cases

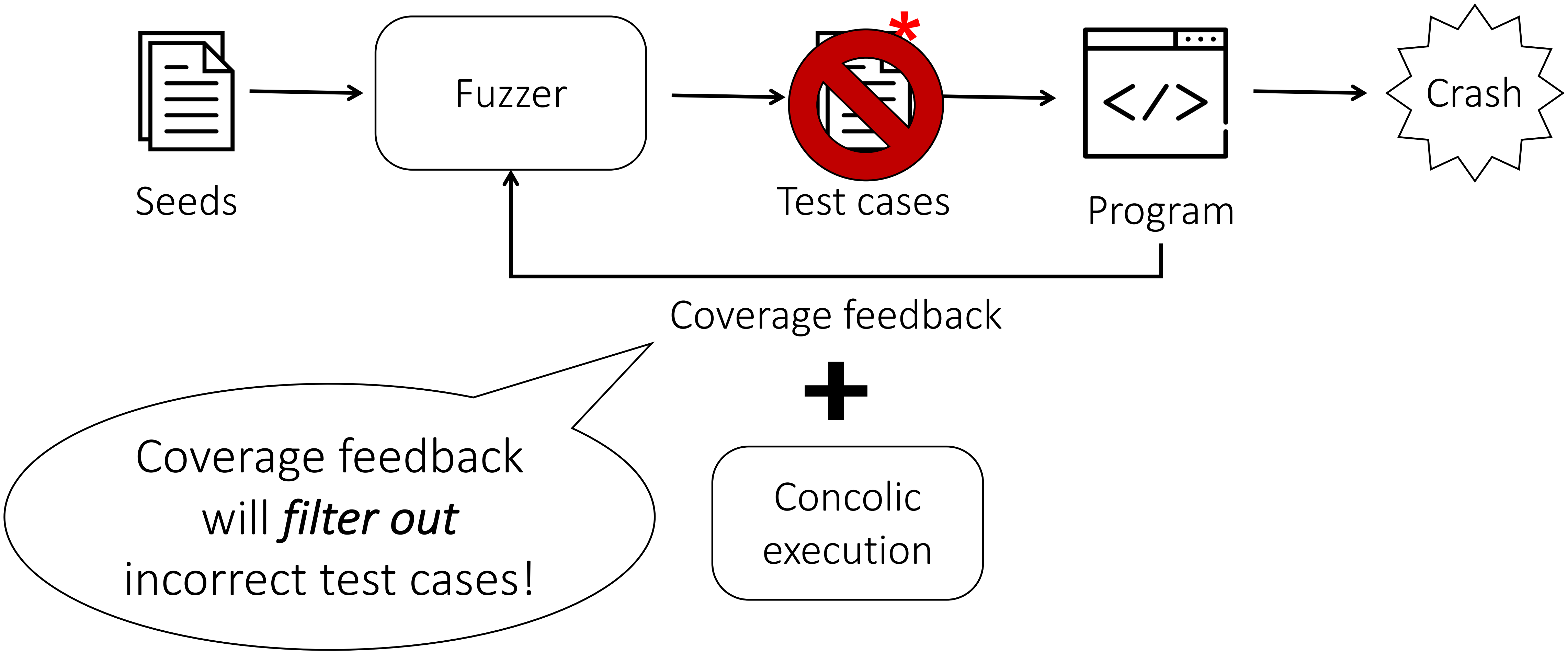
If the test cases are *incorrect*…?

get test cases from
xecution

In hybrid fuzzing, generating incorrect inputs is fine because of coverage-guided fuzzing

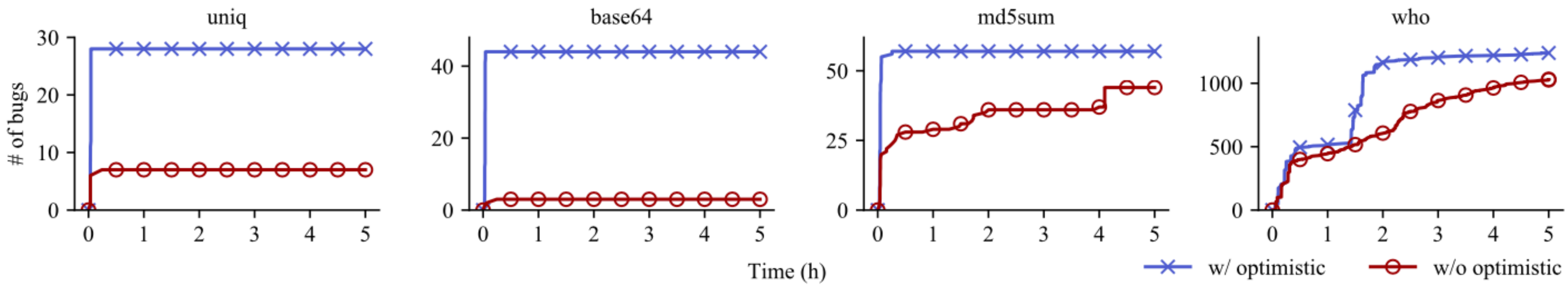


In hybrid fuzzing, generating incorrect inputs is fine because of coverage-guided fuzzing



Optimistic solving helps to find more bugs

- LAVA-M dataset
  - Inject hard-to-reach bugs in real-world applications



Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Remind: Completeness of concolic execution often blocks its further exploration

```
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Limit symbolic execution for repeatedly executed blocks

Remind: Completeness of concolic execution often blocks its further exploration

```
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Limit symbolic execution for repeatedly executed blocks

Can further explore even with such a complicated routine

Remind: Completeness of concolic execution often blocks its further exploration

```c
// 'buf' and 'x' are symbolic
int completeness(char* buf, int x) {
  very_complicated_logic(buf);

  if (x * x == 1234 * 1234)
    crash();
}
```

Limit symbolic execution for repeatedly executed blocks

Can further explore even with such a complicated routine

Incomplete constraints

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because of basic block pruning
```

```
if y == 0xdeadbeef :
```

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

```
if y == 0xdeadbeef :
```

Independent constraints: Use x != 0 in the input

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

```
if y == 0xdeadbeef :
```

Independent constraints: Use x != 0 in the input

```
if x == 0xdeadbeef :
```

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

```
if y == 0xdeadbeef :
```

Independent constraints: Use x != 0 in the input

```
if x == 0xdeadbeef :
```

Subsumed constraints

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

```
if y == 0xdeadbeef :
```

😃

Independent constraints: Use x != 0 in the input

```
if x == 0xdeadbeef :
```

😃

Subsumed constraints

```
if x != 0xdeadbeef :
```

Incomplete constraints are not significant in practice for hybrid fuzzing

```
x = input()
y = input()

// x != 0 is missed because
of basic block pruning
```

```
if y == 0xdeadbeef :
```

Independent constraints: Use x != 0 in the input

```
if x == 0xdeadbeef :
```

Subsumed constraints

```
if x != 0xdeadbeef :
```

Failing...

Evaluating QSYM

- Scaling to real-world software?

- How good is QSYM compared to
  - The state-of-art hybrid fuzzing (Driller)

QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

Program	CVE	Bug Type	Fuzzer
lepton	CVE-2017-8891	Out-of-bounds read	AFL
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz
	Fixed by other patch	NULL dereference	
tcpdump	CVE-2017-11543*	Heap overflow	AFL
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz
libarchive	Wait for patch	NULL dereference	OSS-Fuzz
audiofile	CVE-2017-6836	Heap overflow	AFL
	Wait for patch	Heap overflow × 3	
	Wait for patch	Memory leak	
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz
objdump	CVE-2017-17080	Out-of-bounds read	AFL

QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

Program	CVE	Bug Type	Fuzzer
lepton	CVE-2017-8891	Out-of-bounds read	AFL
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz
	Fixed by other patch	NULL dereference	
tcpdump	CVE-2017-11543*	Heap overflow	AFL
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz
libarchive	Wait for patch	NULL dereference	OSS-Fuzz
audiofile	CVE-2017-6836	Heap overflow	AFL
	Wait for patch	Heap overflow × 3	
	Wait for patch	Memory leak	
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz
objdump	CVE-2017-17080	Out-of-bounds read	AFL

Real-world software

QSYM scales to real-world software

- 13 bugs in real-world software (already tested by fuzzing)

Program	CVE	Bug Type	Fuzzer
lepton	CVE-2017-8891	Out-of-bounds read	AFL
openjpeg	CVE-2017-12878	Heap overflow	OSS-Fuzz
	Fixed by other patch	NULL dereference	
tcpdump	CVE-2017-11543*	Heap overflow	AFL
file	CVE-2017-1000249*	Stack overflow	OSS-Fuzz
libarchive	Wait for patch	NULL dereference	OSS-Fuzz
audiofile	CVE-2017-6836	Heap overflow	AFL
	Wait for patch	Heap overflow × 3	
	Wait for patch	Memory leak	
ffmpeg	CVE-2017-17081	Out-of-bounds read	OSS-Fuzz
objdump	CVE-2017-17080	Out-of-bounds read	AFL

Already heavily fuzzed

QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```
if( !((ox^(ox+dxw))
          | (ox^(ox+dxh))
          | (ox^(ox+dxw+ dxh))
          | (oy^(oy+dyw))
          | (oy^(oy+dyh))
          | (oy^(oy+dyw+ dyh))) >> (16 + shift)
          && !(dxx | dxy | dyx | dyy) & 15
          && !(need_emu&&(h>MAX_H || stride > MAX_STRIDE)))
{ // the bug is here ; }
```

QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```
if( !((c
              | (oy^(oy+dyh))
              | (oy^(oy+dyw+ dyh))) >> (16 + shift)
              && !(dxx | dxy | dyx | dyy) & 15
              && !(need_emu&&(h>MAX_H || stride > MAX_STRIDE)))
{ // the bug is here ; }
```

Cannot be found by cloud fuzzing
(Note: 4 trillion test cases per week)

QSYM can generate test cases that fuzzing is hard to find

- e.g.) ffmpeg: Not reachable by fuzzing

```
if( !((c
    | (oy^(oy+dyh))
                                                RIDE)))
{ // the
```

Cannot be found by cloud fuzzing
(Note: 4 trillion test cases per week)

Found by a *single workstation* using QSYM

QSYM outperforms Driller, the state-of-the-art hybrid fuzzer

QSYM outperforms Driller, the state-of-the-art hybrid fuzzer

- Dataset: 126 CGC binaries
- Compare code coverage achieved by a single run of concolic execution

- QSYM achieved more code coverage in ***104 (82%)*** binaries

Better performance
→ Find deeper code

QSYM is also practically impactful

- e.g., Rode0day: A monthly competition for automatic bug finding tool

Table 2. The overall rankings for top Rode0day competitors after 10 competitions.

Place	Elo score	Team name
1	1,087	afl-lazy
2	1,069	itszn
3	1,027	H3ku
4	1,017	REDQUEEN
5	1,062	NU-AFL-QSYM

Fasano, Andrew, et al. "The Rode0day to Less-Buggy Programs." *IEEE Security & Privacy* (2019)

QSYM is also practically impactful

- e.g., Rode0day: A monthly competition for automatic bug finding tool

Table 2. The overall rankings for top Rode0day competitors after 10 competitions.

Place	Elo score	Team name
1	1,087	afl-lazy
2	1,069	itszn
3	1,027	H3ku
4	1,017	REDQUEEN
5	1,062	NU-AFL-QSYM

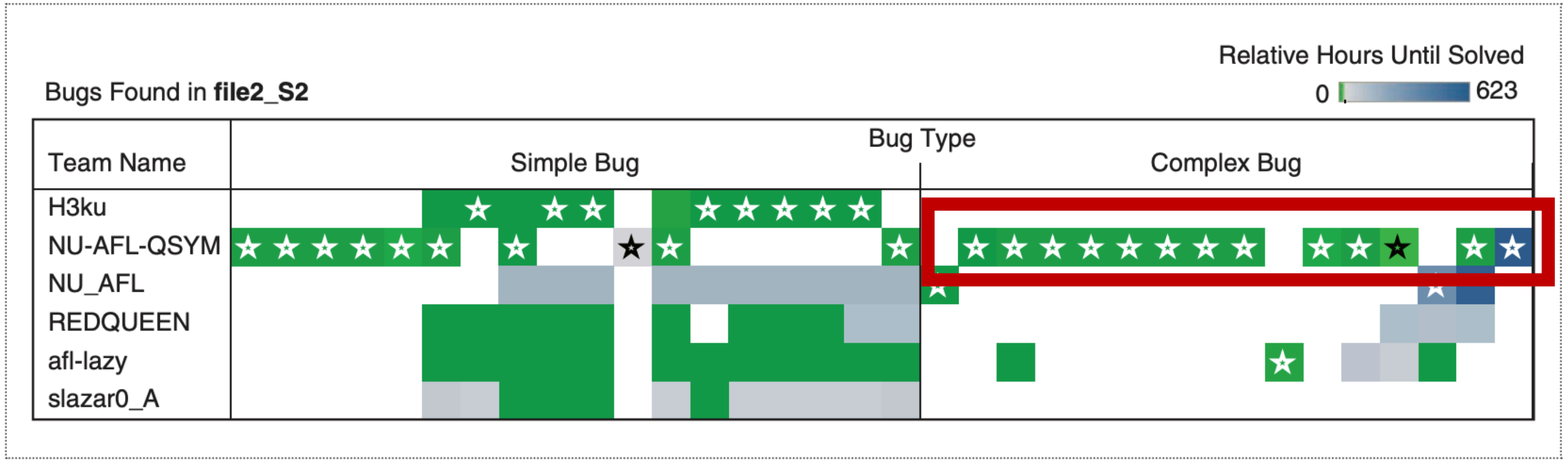


Figure 2. A visualization of when teams found bugs in *file2_S2* during the November 2018 Rode0day. Green indicates the bugs found within 24 h of a team's first score. The stars denote the first team to find a bug.

Fasano, Andrew, et al. "The Rode0day to Less-Buggy Programs." *IEEE Security & Privacy* (2019)

Today's talk

QSYM: A Binary-level Concolic Execution Engine for Hybrid fuzzing

- Binary
- User applications

Hybridra: A Hybrid Fuzzer for Kernel File Systems

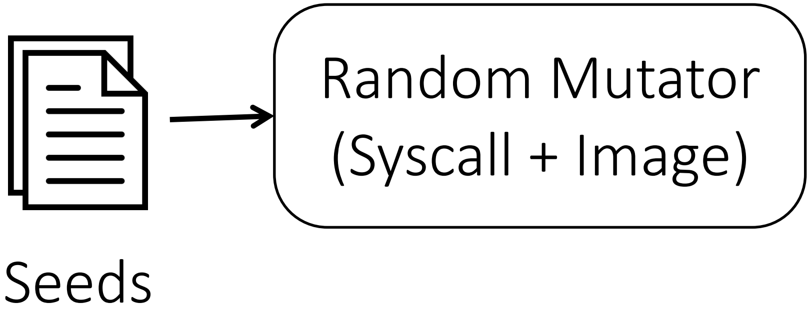
- Source code
- File systems

Hybridra improves Hydra by supporting concolic image mutation

Seeds

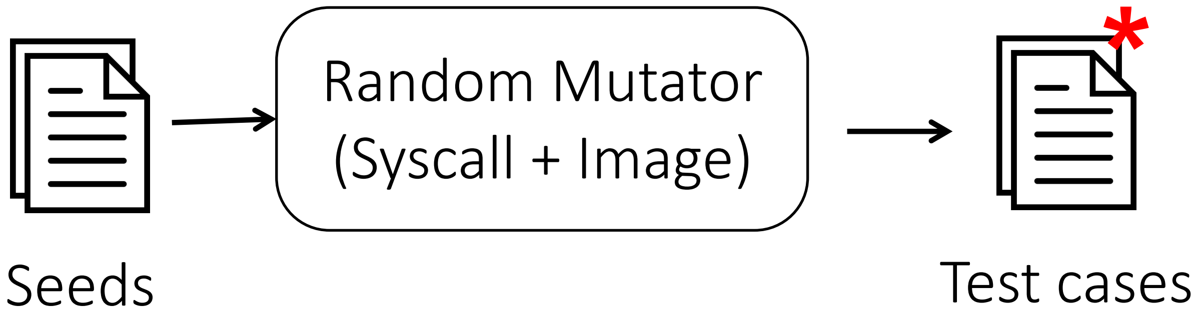
Hydra

Hybridra improves Hydra by supporting concolic image mutation



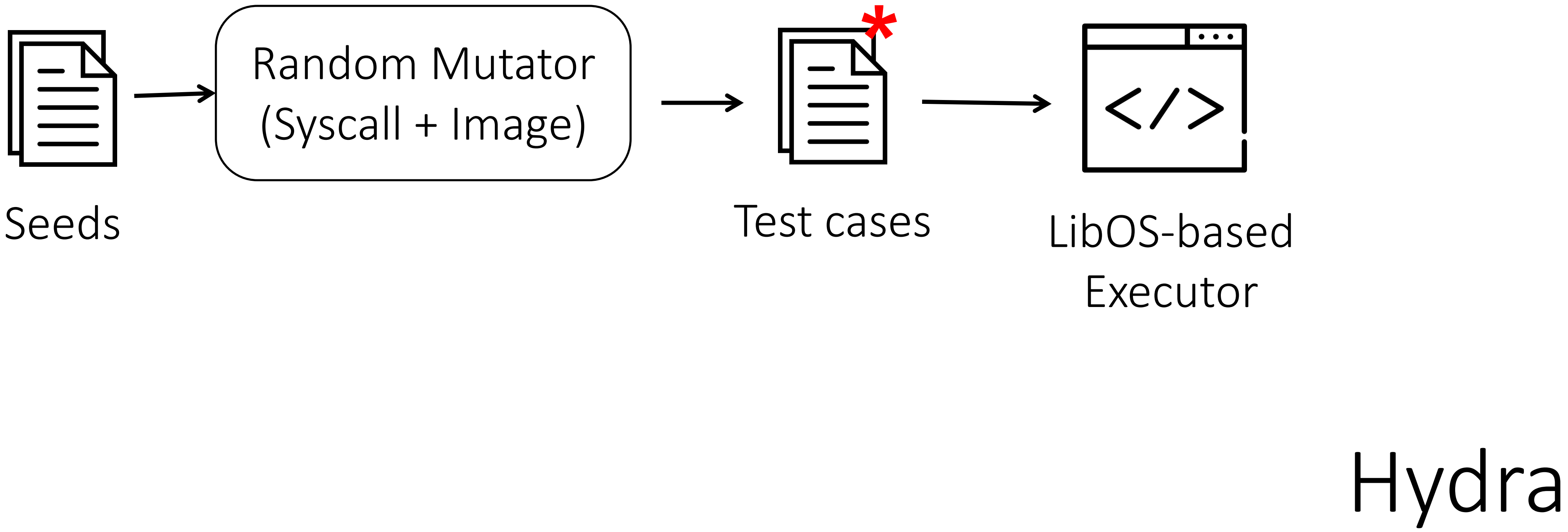
Hydra

Hybridra improves Hydra by supporting concolic image mutation

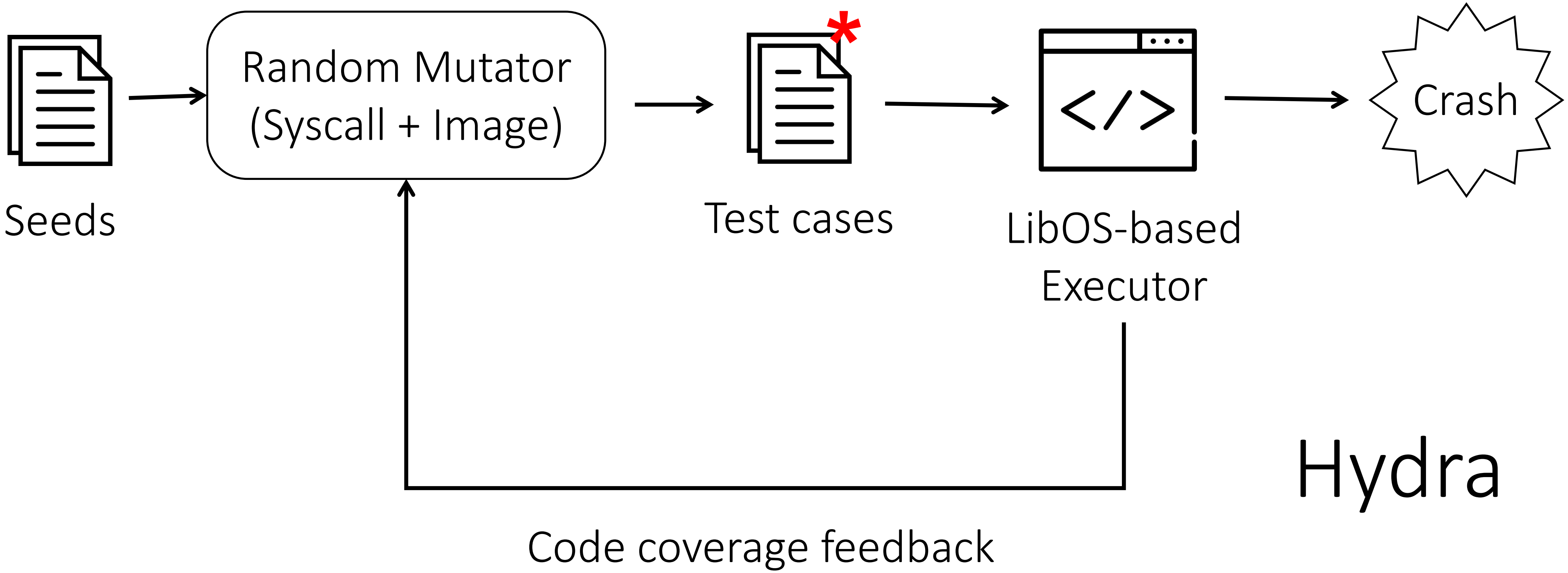


Hydra

Hybridra improves Hydra by supporting concolic image mutation

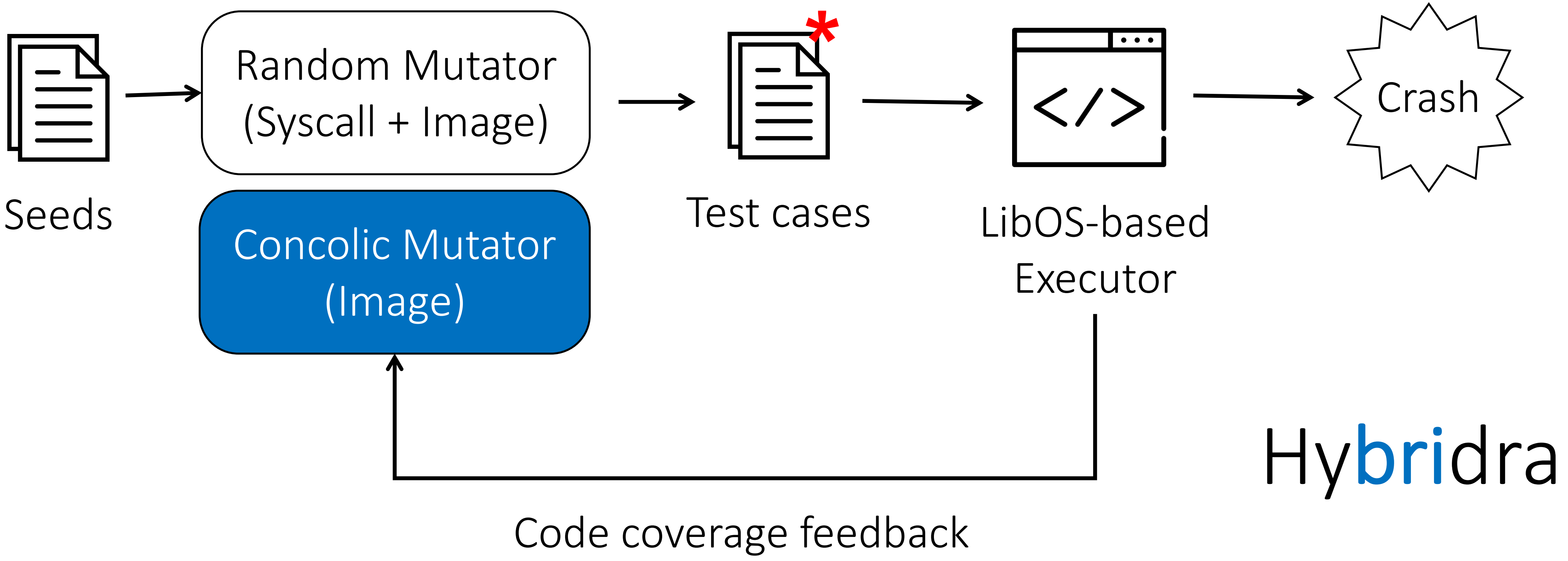


Hybridra improves Hydra by supporting concolic image mutation



Hydra

Hybridra improves Hydra by supporting concolic image mutation



Hybridra: Key ideas

Generating constraints is too slow

→ Compilation-based concolic execution (For source code)

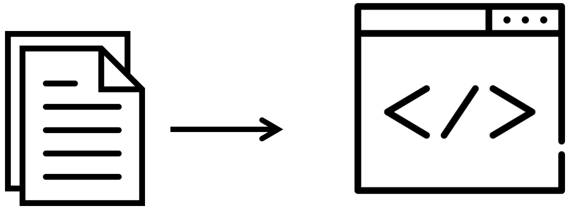
Not effective in generating test cases

→ Staged reduction + Heuristics from QSYM

Design: Concolic Image Mutator

Seeds

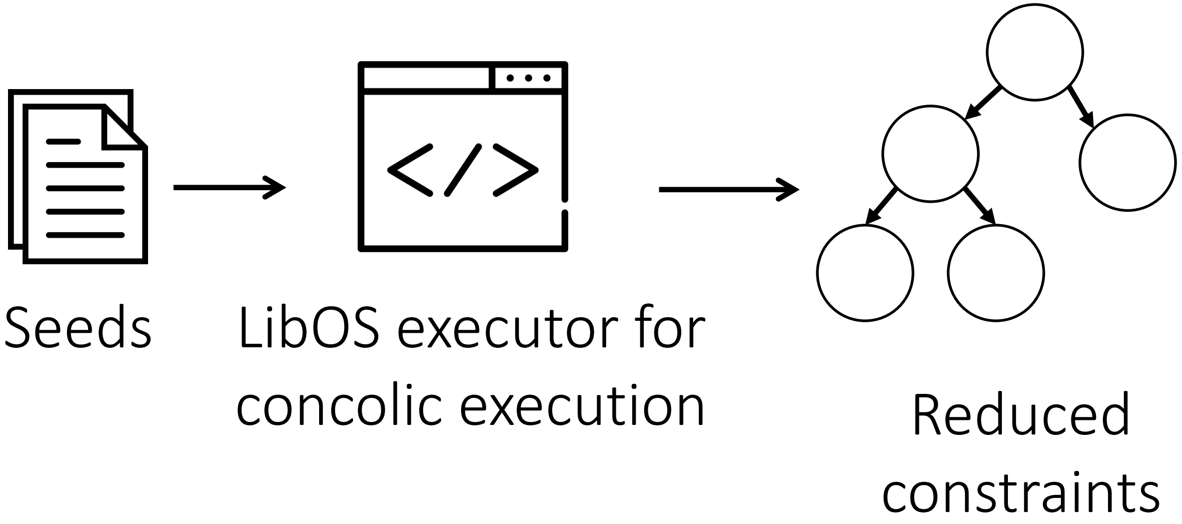
Design: Concolic Image Mutator



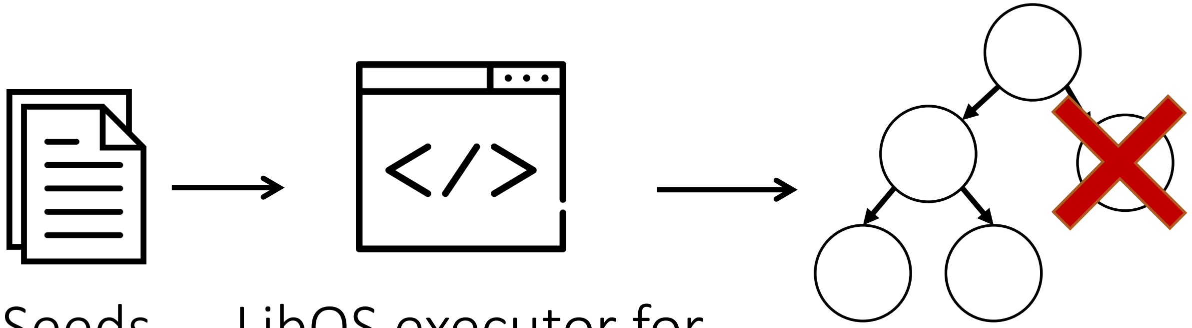
Seeds

LibOS executor for concolic execution

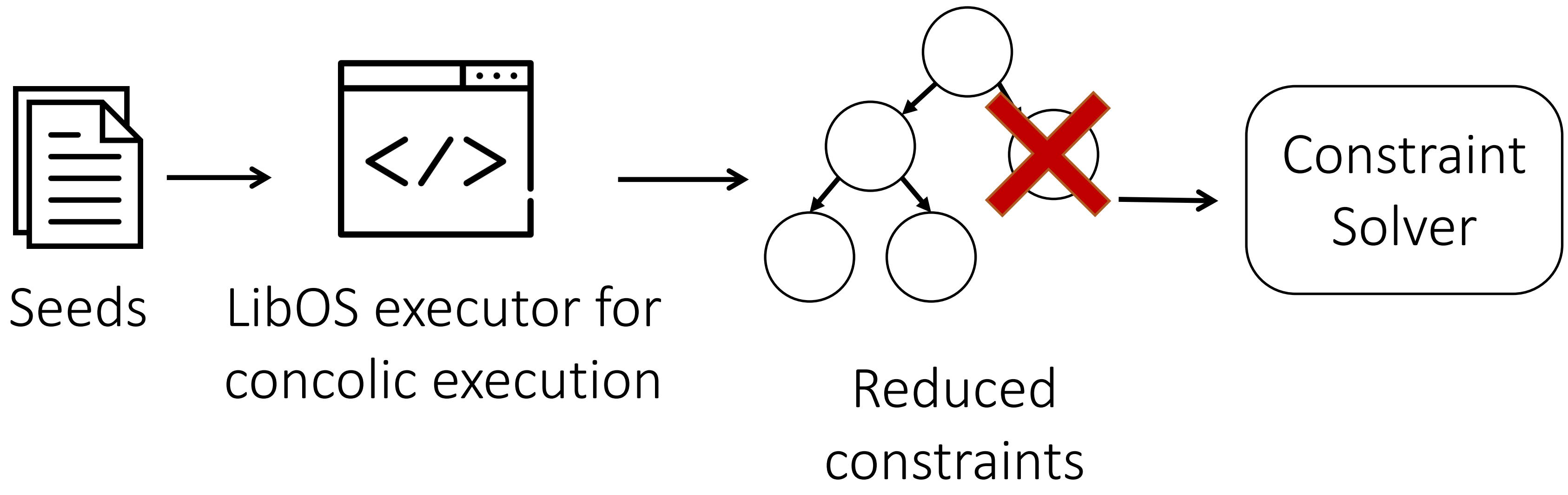
Design: Concolic Image Mutator



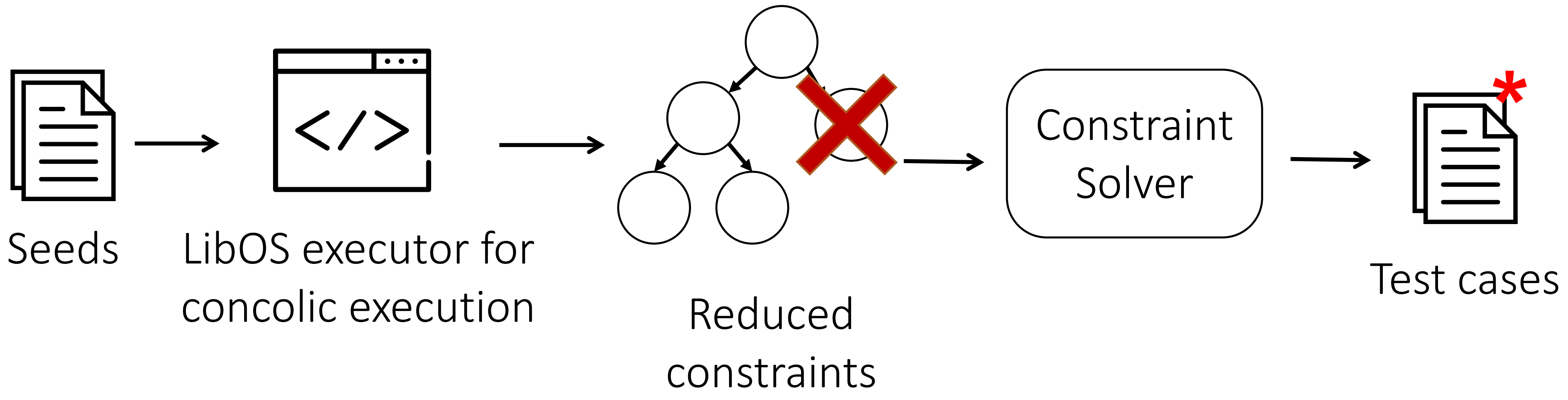
Design: Concolic Image Mutator



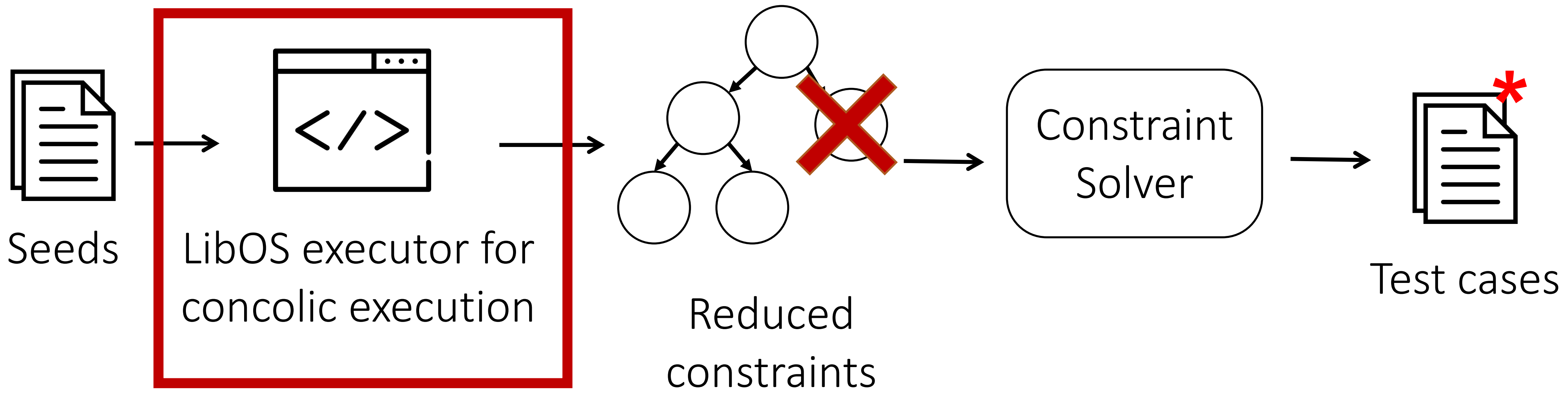
Design: Concolic Image Mutator



Design: Concolic Image Mutator



Design: Concolic Image Mutator



Hybridra utilizes compilation-based concolic execution

Hybridra utilizes compilation-based concolic execution

```
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);

if (c == d) {
   ...
```

Hybridra utilizes compilation-based concolic execution

```
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);

if (c == d) {
    ...
```

Hybridra utilizes compilation-based concolic execution

```
+ Symbol* symA = getSymbol(a);
+ Symbol* symB = getSymbol(b);
+ Symbol* symC = addSymbol(symA, symB);

int c = a + b;

+ Symbol* symD = getSymbol(d);
+ // Make test cases
+ checkEqual(symC, symD);
```

200x performance improvement compared to QSYM
(NOTE: code is required)

Hybridra's compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading

	CUTE	SymCC	Kirenenko
Language	C	LLVM IR	LLVM IR
Memory modeling	Page table	Page table	Shadow memory
Multi-threading			✓

Hybridra's compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading

	CUTE	SymCC	Kirenenko
Language	C	LLVM IR	LLVM IR
Memory modeling	Page table	Page table	Shadow memory
Multi-threading			✓

Hybridra's compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading

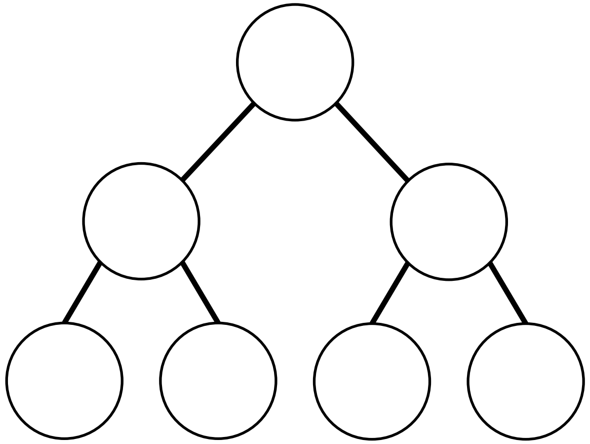
	CUTE	SymCC	Kirenenko
Language	C	LLVM IR	LLVM IR
Memory modeling	Page table	Page table	Shadow memory
Multi-threading			✓

File system	Failure
btrfs	z3 exception
ext4	Deadlock
f2fs	Deadlock
xfs	Deadlock

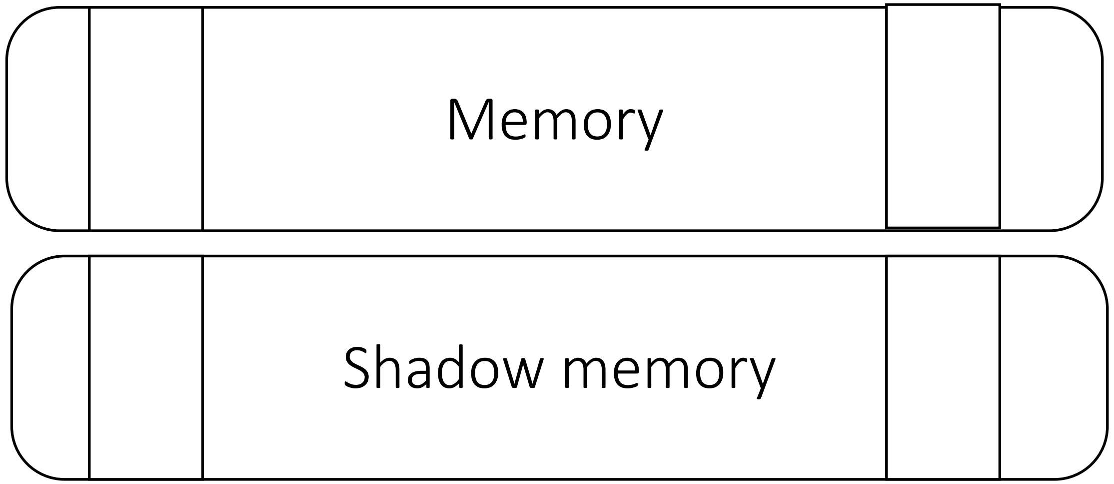
Hybridra's compilation-based concolic execution, Kirenenko, is useful by supporting multi-threading

	CUTE	SymCC	Kirenenko
Language	C	LLVM IR	LLVM IR
Memory modeling	Page table	Page table	Shadow memory
Multi-threading			✓

Comparison: Memory modeling

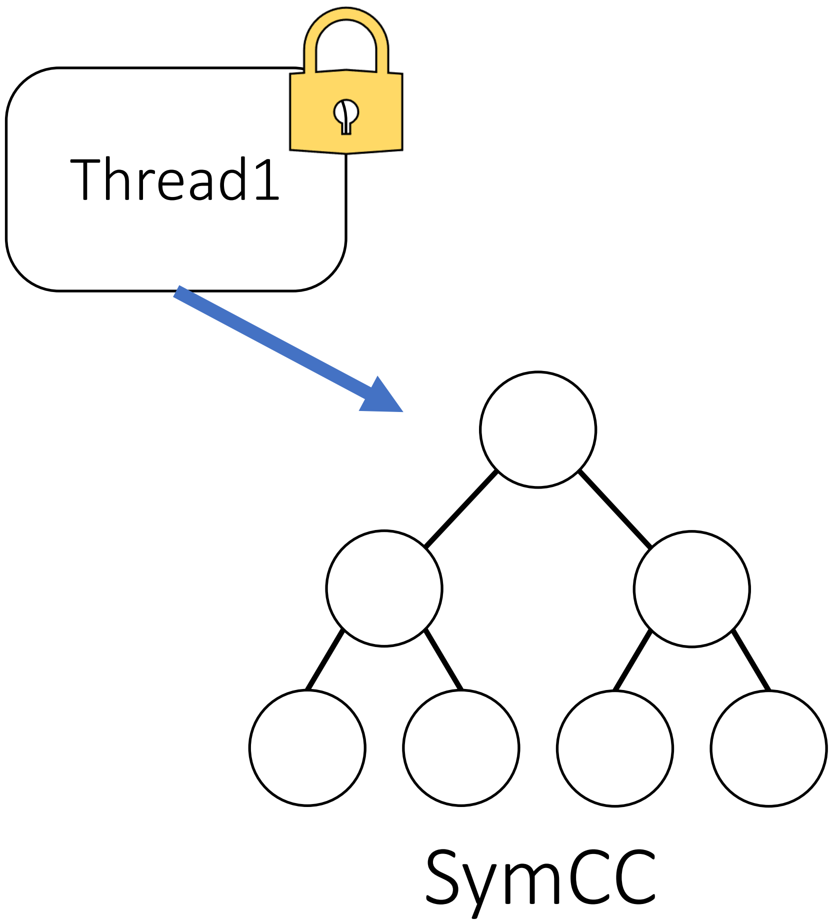


SymCC

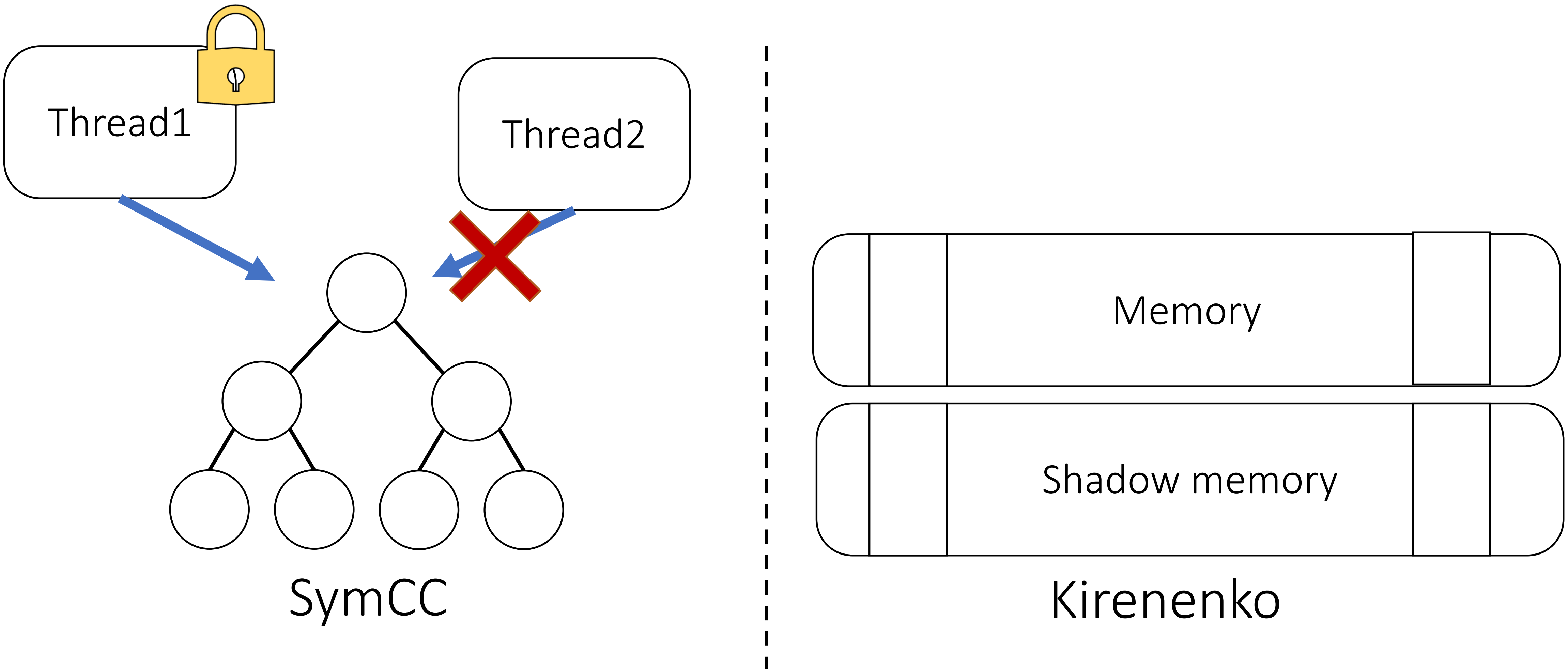


Kirenenko

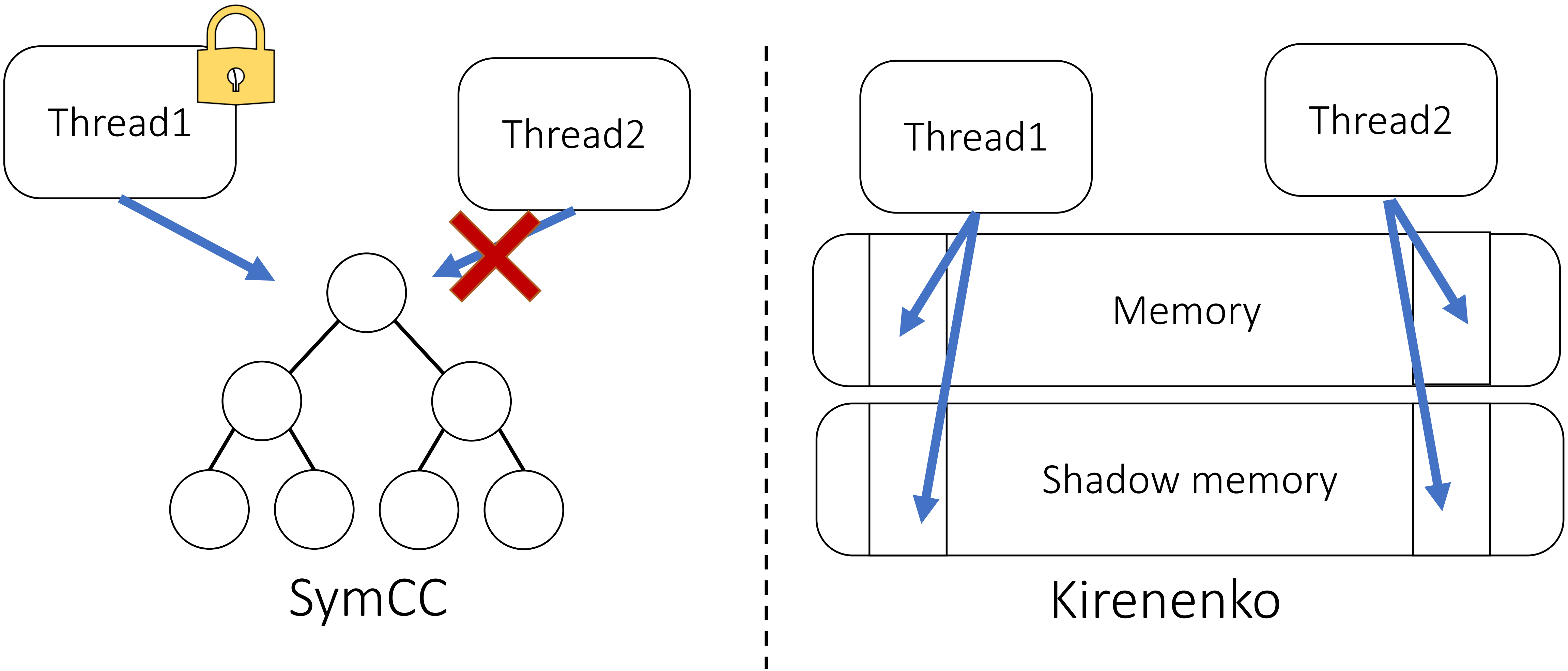
Comparison: Memory modeling



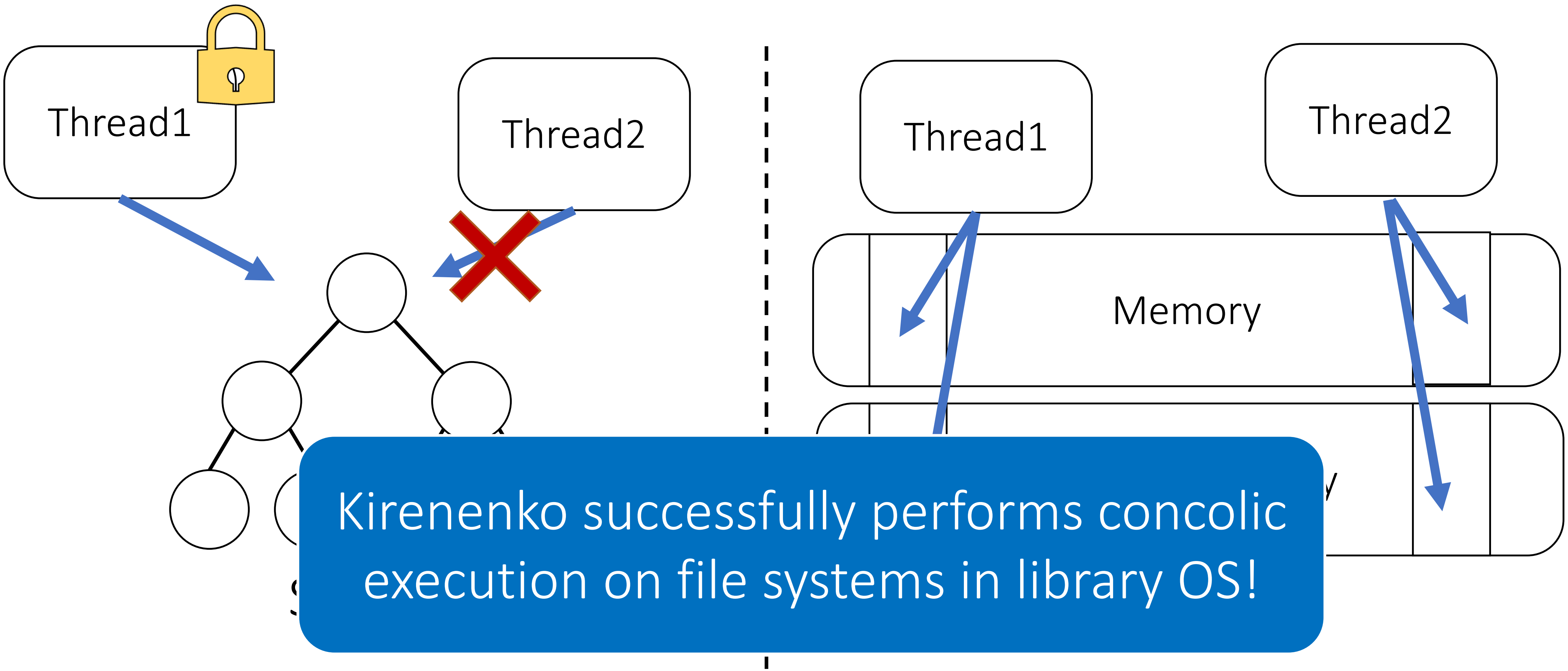
Comparison: Memory modeling



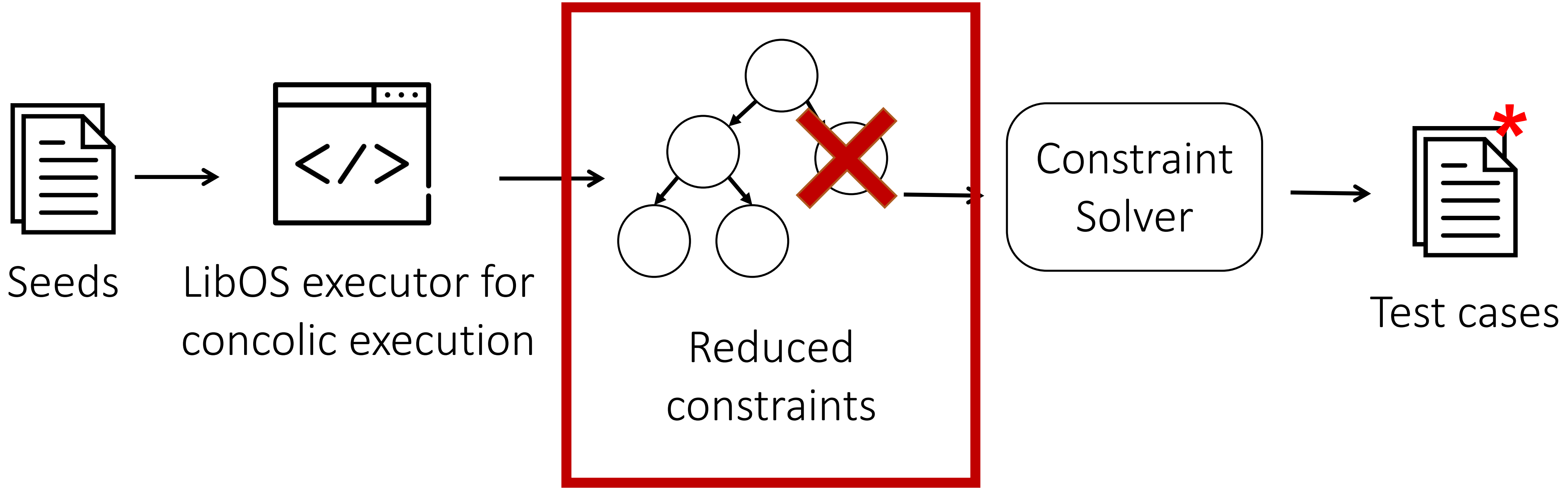
Comparison: Memory modeling



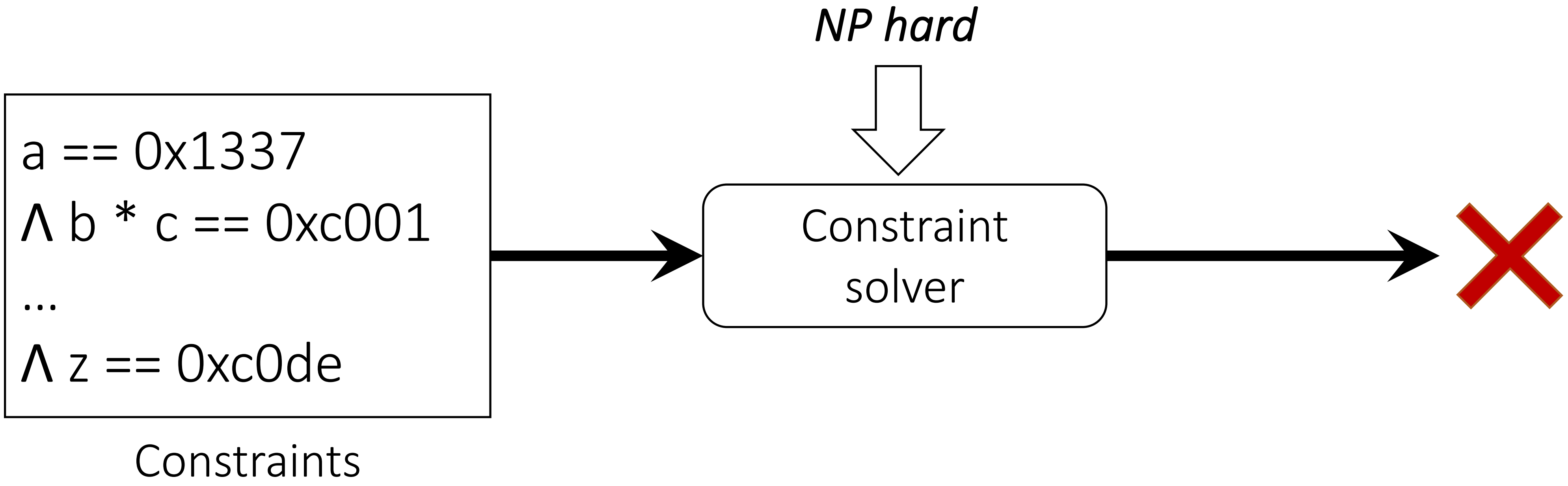
Comparison: Memory modeling



Design: Concolic Image Mutator



Remind: Constraints solving is hard!



Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Fast algorithm

Limited expressiveness

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

Two types of constraints reduction exist

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

No algorithmic improvement

Two types of constraints reduction exist

Never used after
a solver can handle
non-linear operations

∧ b

…

∧ z == 0xc0de

Constraints

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

No algorithmic improvement

Staged reduction: combine both reduction mechanisms

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

1

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

No algorithmic improvement

Staged reduction: combine both reduction mechanisms

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

①

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

②

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

No algorithmic improvement

Staged reduction: combine both reduction mechanisms

③

a == 0x1337
∧ b * c == 0xc001
…
∧ z == 0xc0de

Constraints

①

a == 0x1337
∧ 13 * c == 0xc001
…
∧ z == 0xc0de

Reduction: Complexity
(e.g., Linear reduction)

Fast algorithm

Limited expressiveness

②

~~a == 0x1337~~
∧ b * c == 0xc001
…
∧ z == 0xc0de

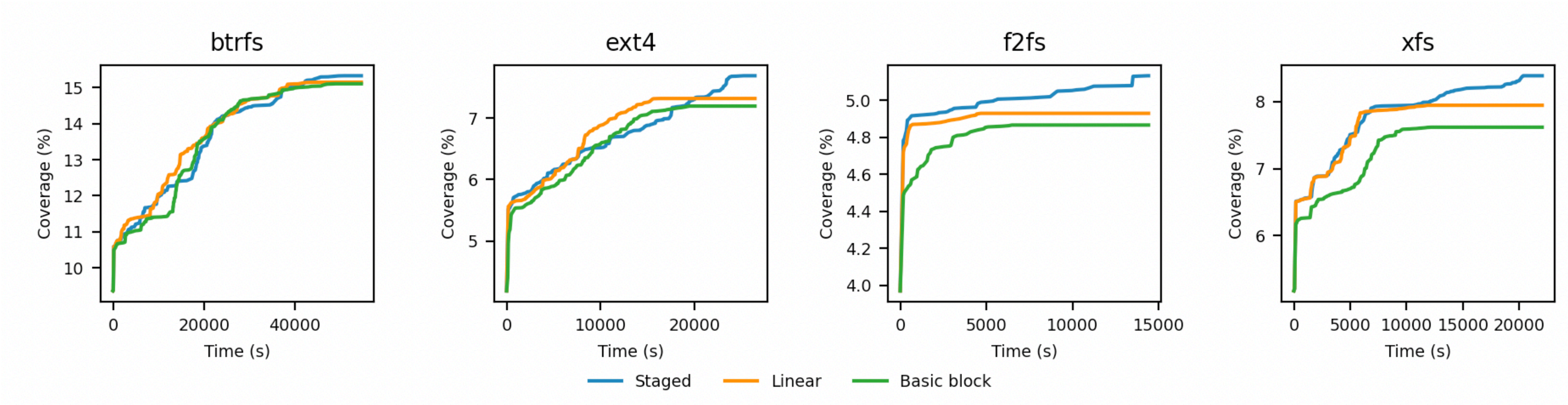
Reduction: # of constraints
(e.g., Basic block pruning)

Expressive

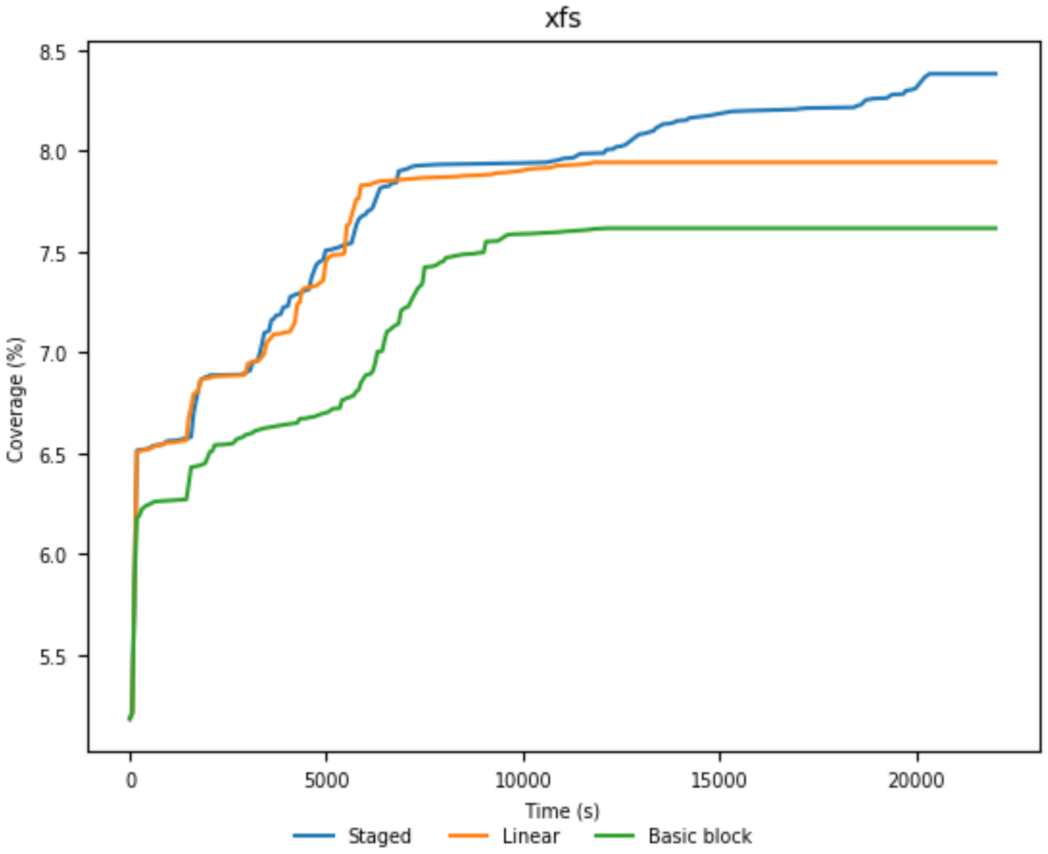
No algorithmic improvement

Staged reduction outperforms each reduction mechanism

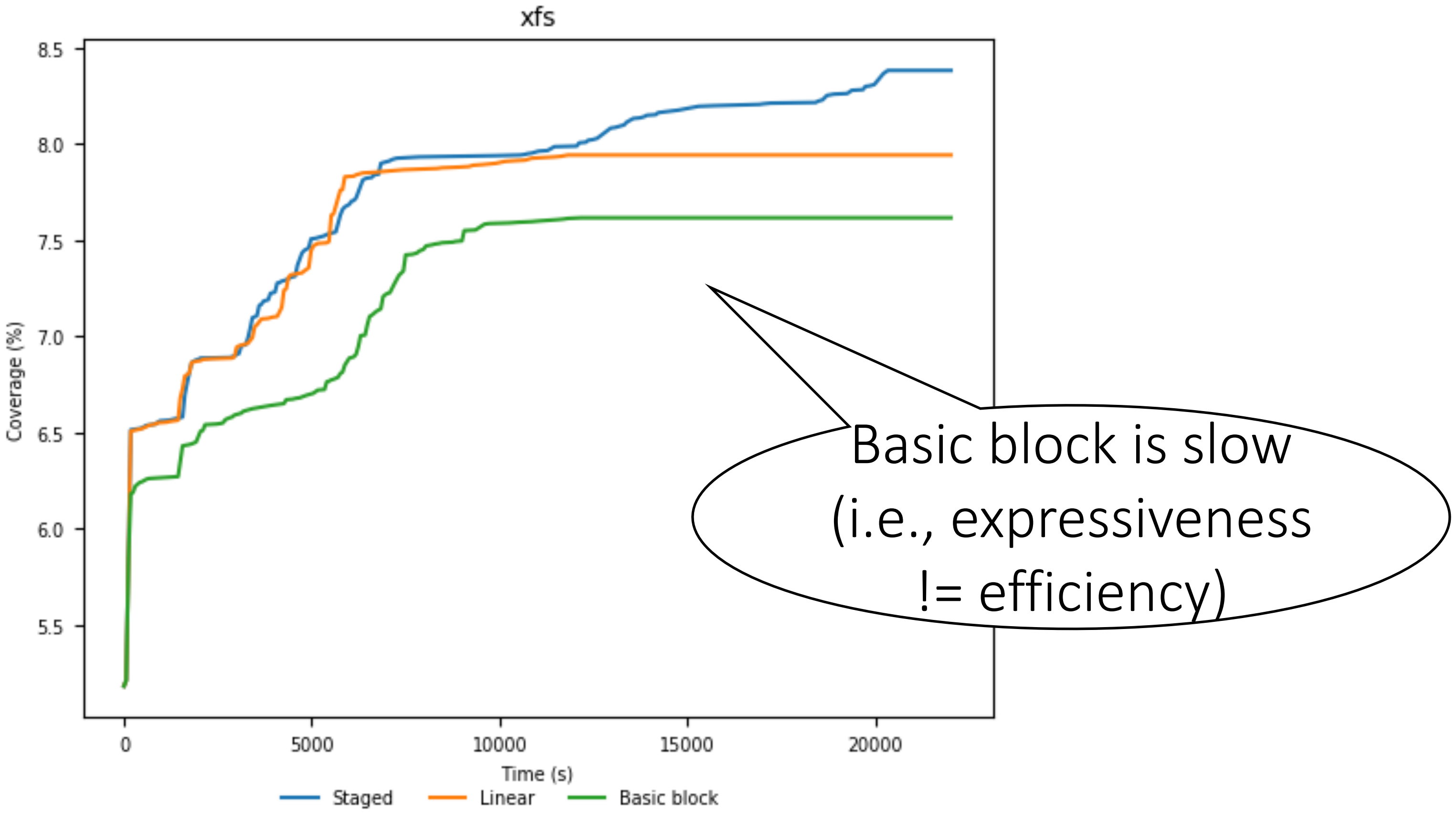
- Setting: Concolic image only, fixed timeout (9 min, 24 hours)



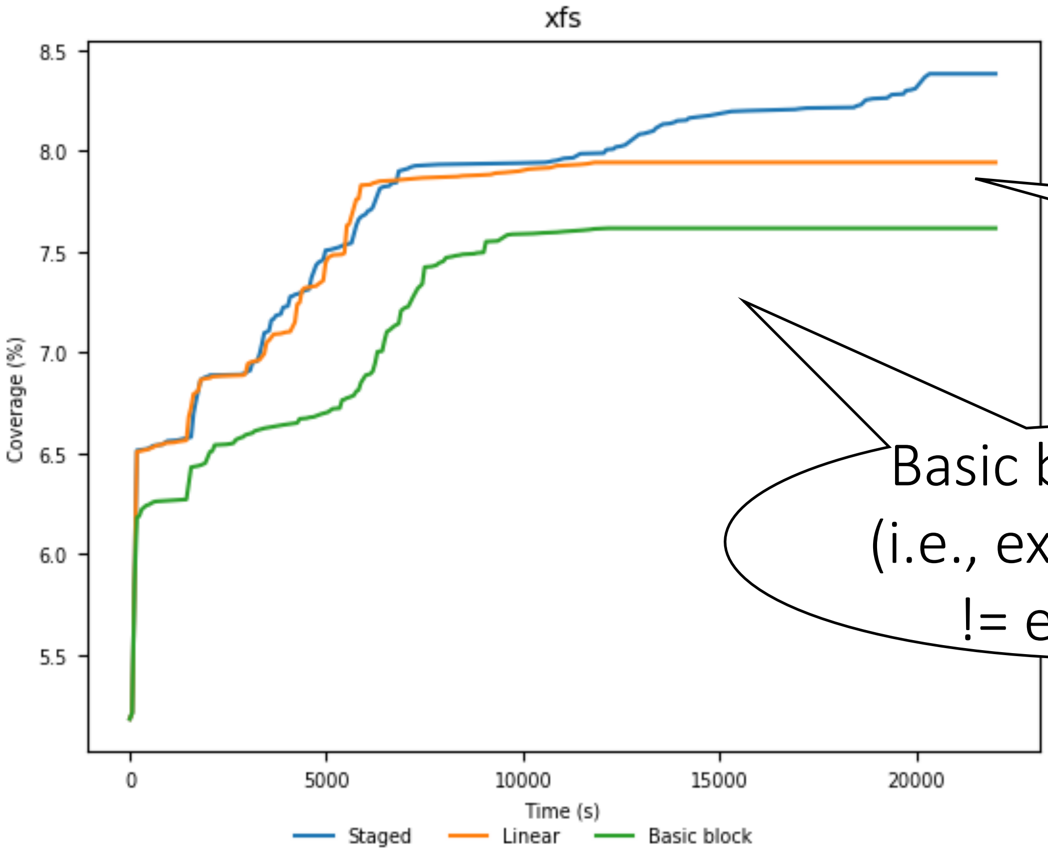
Staged reduction outperforms each reduction mechanism



Staged reduction outperforms each reduction mechanism



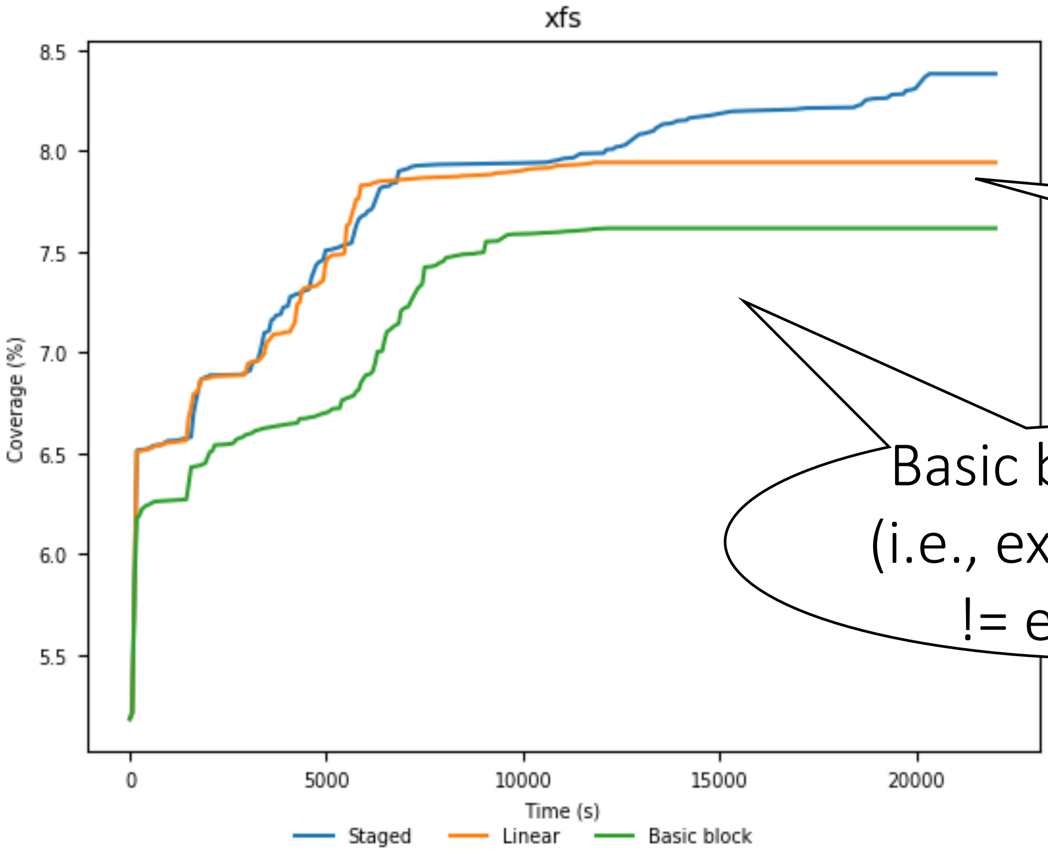
Staged reduction outperforms each reduction mechanism



Linear converges too early due to limited expressiveness

Basic block is slow (i.e., expressiveness != efficiency)

Staged reduction outperforms each reduction mechanism



Linear converges too early due to limited expressiveness

Basic block is slow (i.e., expressiveness != efficiency)

Combining both techniques is useful to achieve higher code coverage!

Evaluation

- Effective to discover new bugs in file systems?
- Outperforms the fuzzing-only solution, Hydra?

Hybridra is effective in finding bugs in file systems

- We fuzz for 2 weeks
  - Each fuzzing takes 24 hours
- Target: Linux v5.3 (LKL), but the latest Linux is v5.8

File system	File	Function	Type	Concolic	New
btrfs	fs/btrfs/extent_io.c	extent_io_tree_panic	Null pointer dereference	✓	✓
	fs/btrfs/free-space-cache.c	tree_insert_offset	BUG()	✓	
	fs/btrfs/extent-tree.c	btrfs_drop_snapshot	BUG()		
	fs/btrfs/extent-tree.c	walk_down_proc	BUG()	✓	
	fs/btrfs/relocation.c	merge_reloc_root	BUG()		
	fs/btrfs/root-tree.c	btrfs_find_root	BUG()	✓	✓
	fs/btrfs/ctree.c	setup_items_for_insert	BUG()		✓
	fs/btrfs/volumes.c	calc_stripe_length	Divide by zero		✓
ext4	fs/ext4/super.c	ext4_clear_journal_err	BUG()		
f2fs	fs/f2fs/segment.c	f2fs_build_segment_manager	Out-of-bounds read		

Hybridra is effective in finding bugs in file systems

- We fuzz for 2 weeks
  - Each fuzzing takes 24 hours
- Target: Linux v5.3 (LKL), but the latest Linux is v5.8

Four new bugs

File system	File	Function	Type	Concolic	New
btrfs	fs/btrfs/extent_io.c	extent_io_tree_panic	Null pointer dereference	✓	✓
	fs/btrfs/free-space-cache.c	tree_insert_offset	BUG()	✓	
	fs/btrfs/extent-tree.c	btrfs_drop_snapshot	BUG()		
	fs/btrfs/extent-tree.c	walk_down_proc	BUG()	✓	
	fs/btrfs/relocation.c	merge_reloc_root	BUG()		
	fs/btrfs/root-tree.c	btrfs_find_root	BUG()	✓	✓
	fs/btrfs/ctree.c	setup_items_for_insert	BUG()		✓
	fs/btrfs/volumes.c	calc_stripe_length	Divide by zero		✓
ext4	fs/ext4/super.c	ext4_clear_journal_err	BUG()		
f2fs	fs/f2fs/segment.c	f2fs_build_segment_manager	Out-of-bounds read		

Hybridra is effective in finding bugs in file systems

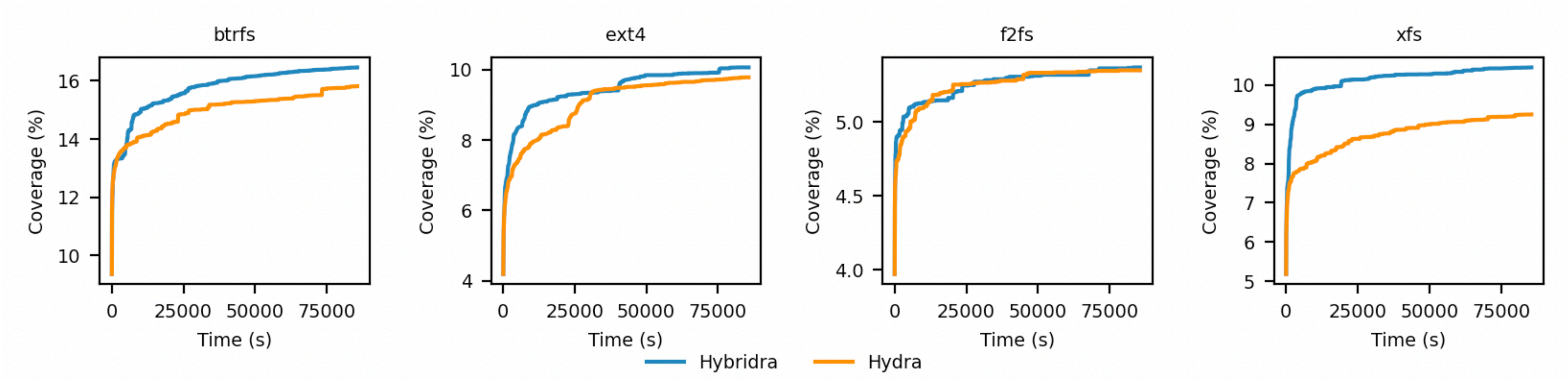
- We fuzz for 2 weeks
  - Each fuzzing takes 24 hours
- Target: Linux v5.3 (LKL), but the latest Linux is v5.8

File system	File	Function	Type	Concolic	New
btrfs	fs/btrfs/extent_io.c	extent_io_tree_panic	Null pointer dereference	✓	✓
	fs/btrfs/free-space-cache.c	tree_insert_offset	BUG()	✓	
	fs/btrfs/extent-tree.c	btrfs_drop_snapshot	BUG()		
	fs/btrfs/extent-tree.c	walk_down_proc	BUG()	✓	
	fs/btrfs/relocation.c	merge_reloc_root	BUG()		
	fs/btrfs/root-tree.c	btrfs_find_root	BUG()	✓	✓
	fs/btrfs/ctree.c	setup_items_for_insert	BU		✓
	fs/btrfs/volumes.c	calc_stripe_l			✓
ext4	fs/ext4/super.c	ext4			
f2fs	fs/f2fs/segment.c	f2f			

Many bugs directly from
concolic execution
e.g., BUG(x != 0);

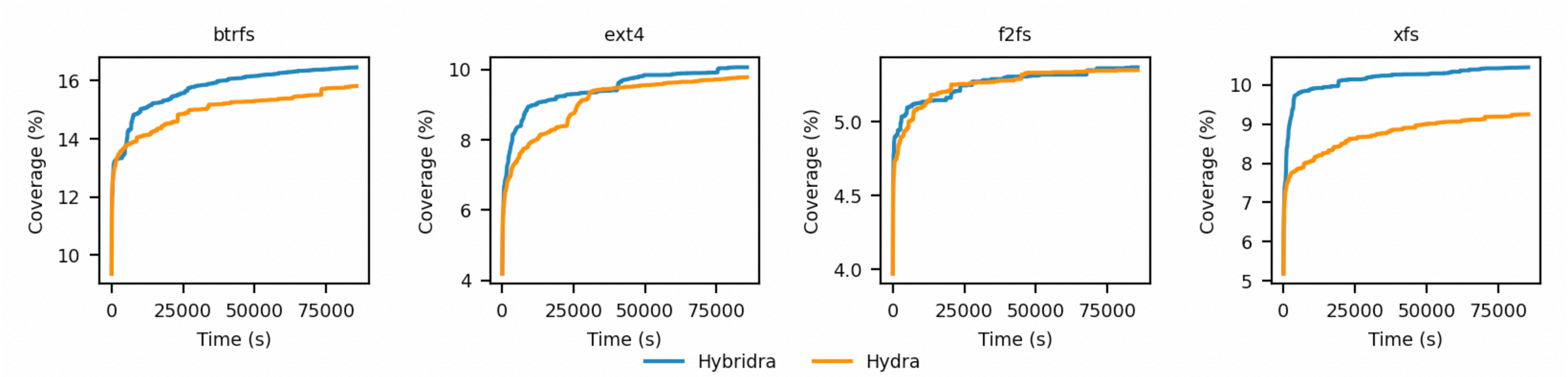
Hybridra outperforms the fuzzing-only approach, Hydra

- Setting: Image only (+ Random), fixed timeout (24 hours)



Hybridra outperforms the fuzzing-only approach, Hydra

- Setting: Image only (+ Random), fixed timeout (24 hours)



Concolic execution can help fuzzing in file systems by discovering interesting test cases!

Discussion & Limitation

Discussion & Limitation

- Apply to other applications
  - Our library OS (LKL) also supports network simulation.
  - Thus, it is possible to extend it to network stacks
  - We can apply other user-mode kernel (e.g., Kunit) to test other features
- Limitations
  - Currently, Hybridra does not support floating point and vector operation
  - The limited number of symbols (2^{30}) because of shadow memory
    - In our evaluation, this is fine for testing file systems

Conclusion

- Designing a concolic executor tailored for hybrid fuzzing is important for scaling hybrid fuzzing to real-world software
  - Systematic approaches for fast symbolic simulation
  - New heuristics for test case generation
- This dissertation demonstrates this idea with
  - QSYM: Hybrid fuzzing for binary-only applications
  - Hybridra: Hybrid fuzzing for file systems

Acknowledgements

- Georgia Tech
  - Taesoo Kim
  - Meng Xu
  - Jinho Jung
  - Wen Xu
  - Soyeon Park
  - Daehee Jang
- Oregon State University
  - Yeongjin Jang
- Virginia Tech
  - Changwoo Min
- Seoul National University
  - Byoungyoung lee
  - Yunheung Paek
- Microsoft Research
  - Sangho Lee
  - Weidong Cui
  - Xinyang Ge
  - Ben Niu
- University of Michigan
  - Baris Kasikci
  - Upamanyu Sharma
- Arizona State University
  - Ryuou Wang
- Google
  - Chanil Cheon
- Facebook
  - Dhaval Kapil
- University of Pennsylvania
  - Xujie Si
  - Mayur Naik
- UNIST
  - Hyungon Moon
- NSRI
  - Su yong Kim
- KAIST
  - Yongdae Kim
  - Kyoungsoo Park
  - Yung Yi

Thank you!